版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！

VISIT…

LANZAROTE Caliente.com

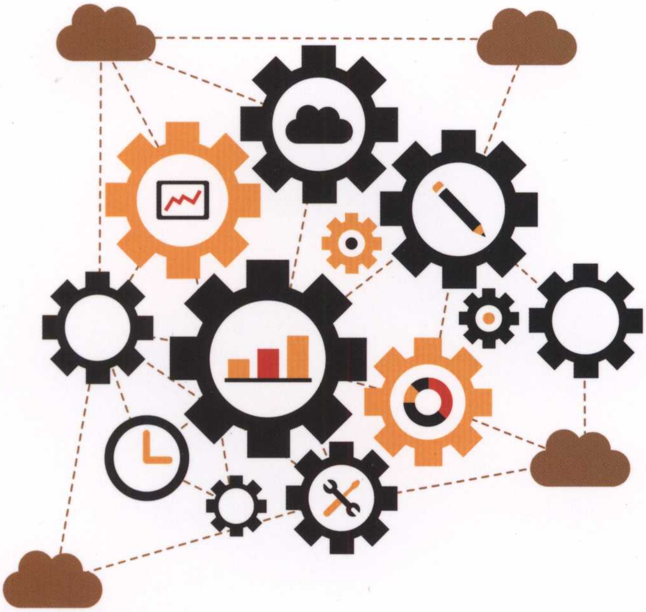
Broadview®
www.broadview.com.cn

基于Spring Cloud Camden SR4 Docker 1.13.0

Spring Cloud与Docker

微服务架构实战

周立 著



中国工信出版集团

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

周立 ▶

Spring Cloud 中国社区联合发起人。拥有近 7 年的软件系统开发经验，多年系统架构经验，对 Spring Cloud、微服务、持续集成、持续交付有一定见地。

他热爱技术交流，曾代表公司参加全球微服务架构高峰论坛、QCon 等技术沙龙；拥抱开源，在 GitHub 与 Git@OSC 上开源多个项目，例如开源电子书《使用 Spring Cloud 与 Docker 实战微服务》等，并获得了开源中国的推荐。

作者博客：http://itmuch.com，定期分享 Spring Cloud 相关文章。读者可扫码关注 Spring Cloud 中国社区公众号以及作者公众号。

Spring Cloud与Docker

微服务架构实战

周立 著

電子工業出版社

Publishing House of Electronics Industry

北京•BEIJING

内容简介

作为一部帮助大家实现微服务架构落地的作品，本书基于 Spring Cloud Camden SR4 Docker 1.13.0，覆盖了微服务理论、微服务开发框架（Spring Cloud）以及运行平台（Docker）三大主题。全书可分为三部分，第 1 章对微服务架构进行了系统的介绍；第 2~11 章使用 Spring Cloud 开发框架编写了一个“电影售票系统”；第 12~14 章则讲解了如何将微服务应用运行在 Docker 之上。全书 Demo 驱动学习，以连贯的场景、具体的代码示例来引导读者学习相关知识，最终使用特定的技术栈实现微服务架构的落地。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

Spring Cloud 与 Docker 微服务架构实战 / 周立著.—北京：电子工业出版社，2017.5

ISBN 978-7-121-31271-7

I. ①S… II. ①周… III. ①互联网络 – 网络服务器 IV. ①TP368.5

中国版本图书馆 CIP 数据核字（2017）第 070057 号

策划编辑：张春雨

责任编辑：徐津平

印　　刷：北京天宇星印刷厂

装　　订：北京天宇星印刷厂

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱　　邮编：100036

开　　本：787×980　1/16　印张：17　字数：342.18 千字

版　　次：2017 年 5 月第 1 版

印　　次：2017 年 5 月第 1 次印刷

定　　价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819　faq@phei.com.cn。

序 1

2016 年国庆假期之后，我所在的公司因为业务需要，想搭建一个 API 网关来综合治理已有业务调用服务（我司之前采用的是当当的 Dubbo 扩展框架 Dubbox）。前期，我和同事们在技术选型环节，讨论了诸多目前比较红火的技术框架和工具。最后达成一致，采用微服务来重构和调整原先这些 Dubbox 服务，并决定使用 Spring Cloud（以下简称 sc）来实现 API 网关，争取在 2017 年能顺利平滑地从 Dubbox 过度到 sc。而具体的 API 网关 demo 研发工作就落实到了我这里。

在开始研发工作之前，我参阅了包括官网在内很多 sc 研发资料，也去全球最大的同性技术交友网站 GitHub 上找了很多代码来仔细研读。但感觉老外的这些 Guide（指南）总是讲得不是很通透。也许是有些概念他们觉得太基础了，就直接略过不表。因此我也感到很迷茫，老是问自己，到底应该如何去实现这个 API 网关，完成公司指派给我的研发任务呢？

幸好，某一天我看到《Spring Cloud 与 Docker 实战微服务》这本开源书。根据书中例子，我几乎没有费什么大工夫就搭建了一个 API 网关的 demo。甚至其中某些讲解点，看了之后能让我一下子恍然大悟，回头再看那些老外的 Guide，我终于明白了其中的“奥秘”。我真的非常感激这位开源书作者，他深入浅出地将 sc 所涉及的各种知识点和工具的使用做了完整和详细的叙述。从此，我也记住了此书作者的网名 itmuch。

几天后，我将 demo 做了细化和扩展，并在 oschina 的码云网站上开源分享了出去（具体网址见 *http://git.oschina.net/darkranger/spring-cloud-books*）。而无巧不巧，itmuch 居然在我项目下的评论区留言了。经过加 QQ、加微信等一系列同性技术交友过程（你们懂的）取得了联系，也终于知道了这位 itmuch 的真名，那就是此书作者周立同学。在闲聊过程中，他透露了自己正在以那本《Spring Cloud 与 Docker 实战微服务》开源书为基础，继续扩展和具体深入 sc 这套微服务开发体系所包含的所有技术点，准备出版成册，让更多的朋友和企业能学习和借鉴 sc 这套东西开发符合自己业务场景的微服务框架。并邀请我为新书做校对和修正工作。正巧，我也越来越喜欢钻研 sc，也希望对自己碰到的一些问题向他指教，所以就答应了下来。这其中的过程真的一言难叙，总算最后我也不辱使命地完成了这本书的校对工作。

我可以很负责地说，本书是周立同学本人在工作和学习 sc 后总结出的精华，书中每段代码、每个字都是他自己写的，绝无任何抄袭之举。完全可以说是一个努力勤奋、能独立思考、认真做事的同学的良心之作。这样的“业界良心”在如今这个充满浮躁的社会中已不多见。希望有更多的读者能珍惜此书，感谢周立同学给我们的帮助。除此之外，我也希望读者能在阅读完此书后，可以自己写点代码，亲身去实践一把，感受 sc 的精妙之处。

最后，在仓促完成本文之前，值此新春佳节之际，我也祝大家新年快乐，家庭安康，财源滚滚，爱情事业双丰收。

2017 年 1 月 25 日

农历丙申年腊月二十八

吴峻申　青客机器人有限公司架构师

序 2

2013 年，我在 EMC 听了一个关于 Docker 与测试的分享，才第一次近距离认识 Docker。在 2014 年底时，在项目上开始接触 Docker。2015 年上半年，我读了两本书：*The Phoenix Project* 和 *Migrating to Cloud-Native Application Architectures*。这两本书让我对 DevOps、微服务和云原生架构有了初步的认识。

2015 年 9 月，我以首席架构师的身份加入麻袋理财，当时第一件事情就是借助 DaoCloud 在公司内部推行基于 Docker 的基础落地的方案。花了三个月，一个简易的方案就已经可以正常运作。但是在这个过程中，却发现和应用的契合度不是太高，需要对应用的架构做改造。

2016 年年初，当时正好有一个项目要做 2.0，之前是一个典型的单体应用（使用 Spring MVC），这次准备做微服务改造，以满足业务对技术快速迭代、横向扩展的要求。我当时对 Spring Boot 和 Spring Cloud 已经有所耳闻，但是还停留于 Demo 的地步。正好借着这个机会，准备推广 Spring Boot。之后有个全新的项目，我们完全按照微服务架构，使用 Spring Boot 和 Cloud 进行开发，并采用 CI/CD 自动化流程和容器化部署。

2016 年 10 月份时，一次偶然的计划，Spring Cloud 中国社区的许进找到了我，让我把团队在实践过程中的经验总结在社区做了分享，从而认识了本书的作者周立。当时周立正好在写一本书，他希望我能够帮他进行 review，我就欣然答应了。

看到了书的标题《Spring Cloud 与 Docker 微服务实战》，这不就是我一直在做的工作吗？于是我连夜把这本书读了一遍，感觉相见恨晚，如果一年前有这本书，那我就可以少走很多弯路了。

本书用一个例子贯穿始终，讲解了 Spring Cloud 的经典组件、微服务架构，以及与 Docker 的集成。书中提供了详细的代码，可以让读者在了解基础概念的同时，可以马上脚踏实地地撸起袖子写代码。

王天青　DaoCloud 首席架构师

2017 年 3 月

序 3

最近几年，微服务的概念非常火爆，由于它确实能解决传统单体应用所带来的种种问题（比如代码可维护性低、部署不灵活、不够稳定、不易扩展，等等），所以大家对“如何成功实施微服务架构”越来越感兴趣。在 Java 技术栈中，Spring Cloud 独树一帜，提供了一整套微服务解决方案，它基于 Spring Boot 而构建，延续了 Spring 体系一贯的“简单可依赖”，但是由于微服务本身涉及的技术或概念比较广，所以在正式“入坑”之前，最好能有一本实战性强的书籍作为参考。但是很遗憾，Spring Cloud 太新了，国内几乎没有一本完整讲解其用法的新书。

今年年初，我偶然得知周立兄在编写 Spring Cloud 相关的书籍，感到非常惊喜，在和他交流的过程中，我能感觉到他对技术的把控力以及对知识分享的热情！阅读这本书的过程是非常愉悦的，不仅仅是因为它结构之清晰，文风之流畅，更重要的是实战性极强，相信大家能在本书的指导下顺利地基于 Spring Cloud & Docker 打造出自己的微服务应用。

杜云飞　上海小虫数据技术合伙人，风控大数据负责人

序4

在 Spring 尚未出现的“蛮荒”时代，Java 程序员们还在迷茫地创造着各种“语法糖”来试图提高生产效率。然而无论怎么努力，Java 语言仍被许多人冠以“裹脚布”的名号——毕竟你一不小心就会把它写得又臭又长。

随着 Spring 体系的出现与逐步完善，似乎有一种经历着 Java 工业革命的感觉。的确，任何事物都各有利弊，但我仍然想说，Spring 团队给 Java 程序员们带来了春天（就像它的名字一样），它神奇地把“裹脚布”变成了“丝绸”，因为它最大的特质可以用两个字来形容——优雅。相信你如果使用过 Spring Framework、Spring MVC、Spring Data、Spring Boot 或 Spring Cloud 等一系列框架，并研读过它们的源代码，就一定能够体会到“优雅”二字的含义。

尽管 Spring 家族拥有如此多而美好的大块“语法糖”，但它们过去在国内的传播似乎都不怎么顺利。我经常说，国内对新技术的广泛应用一般比国外要晚三到五年，无论后端、前端还是架构理念。这是许多因素导致的，比如信息闭塞、语言不通，甚至固步自封。我相信随着国内互联网人才越来越多，新技术应用的延时一定会越来越短。或许很多人为了旧系统的稳定而不愿升级，这可以理解，但我希望人们可以拥抱新的事物，而不是排斥。现如今微服务架构理念兴起，人们急需一个快捷、稳定、一站式的分布式微服务解决方案，Spring Cloud 正是为此而诞生。可国内熟知 Spring Cloud 的人目前仍寥寥无几，大部分人从未听说过，想要学习的人不知从何开始，对官方的英文文档也一知半解。人们需要一本能把他们领进 Spring Cloud 这扇门的“红宝书”，这便是本书的目的，也是本书作者周立的初衷——希望能够为减少国内新技术的延时而出一份力。

我与周立在 2016 年相识，在短暂的交流后我们都产生了相见恨晚的感觉。遇见志同道合的人并不容易，我们的技术理念很相似。他有着对技术的热忱、灵活的头脑，以及开源分享技术的无私精神，正是这股精神促使他做了许多分享技术的事情，并且编写了这本书（相信我，写书并不赚钱）。我十分欣赏周立身上的这些特质，因此当他跟我提到想出书并找我帮忙时，我毫不犹豫地答应了他。我相信他未来一定能够成为某一技术领域的专家，这是他的目标，他也具备这样的潜质。

本书的切入点非常好，它并不纠结于冗长的源码解读或原理解释，而是更多地注重实战，这在如今互联网爆炸式发展的时代相当重要。现在人们更倾向于使用敏捷开发尽快做出产品来进行试错，并在后续版本中快速迭代。因此本书的实战经验在软件工程层面上会给予阅读者很大提升，它可以让你更快地搭建分布式微服务架构，然后把精力留在编写业务逻辑上，提高你的生产力，并最终做出更好的产品——这也是 Spring 团队一直希望达到的效果。

现在，让我们随本书进入 Spring Cloud 的世界，一起感受它的优雅吧！

张英磊　云账房 CTO

2017 年 3 月 29 日

前言

随着业务的发展，笔者当时所在公司的项目越来越臃肿。随着代码的堆砌，项目变得越来越复杂、开发效率越来越低、越来越难以维护，小伙伴们苦不堪言，毫无幸福感可言。我们迫切需要能够解放生产力、放飞小伙伴的“良药”，于是，微服务进入视野。然而，微服务究竟是什么，众说纷纭，没有人能说清楚什么是微服务。不仅如此，大家对微服务的态度也是泾渭分明，吹捧者、贬低者比比皆是，在笔者的 QQ 群、微信群中硝烟四起。笔者参加了不少交流会，感觉许多分享常常停留在理论阶段。一场会下来，觉得似乎懂了，却苦于没有对应的技术栈去实现这些理论。

Docker、Jenkins 等工具笔者均有涉猎，然而使用什么技术栈去实践微服务架构，在很长时间内都是笔者心中的疑问。

2015 年中，笔者偶然在 GitHub 上看到一个名为 Spring Cloud 的框架，它基于 Spring Boot，配置简单、设计优雅，并且大多组件经过了生产环境的考验。笔者花 1 个月左右的时间详细研究了 Spring Cloud 的相关组件后，体会更深。然而，技术选型必须要进行客观、多维度、全方位的分析，而不应由我个人的主观意见作为决定因素。**文档丰富程度、社区活跃度、技术栈生态、开发效率、运行效率、成功案例**等，都是我们选型的重要因素。经过调研，其他几点都很 OK，只缺成功案例——在当时，国内几乎没什么成功案例，甚至连中文的博客、相关资料都没有。

这让笔者陷入两难，在这一过程中，公司一边继续使用阿里巴巴开源的 Dubbo（Dubbo 虽然在国内非常流行，但毕竟有段时间没有维护了，开源生态也不是很好），一边在笔者的组织下进行一些 Spring Cloud 相关的技术分享。一方面是希望借此开拓小伙伴们的视野，另一方面也希望能将两者相互印证，看能否在现有平台上借鉴 Spring Cloud 的设计或使用其部分组件。

2016 年 8 月，笔者有幸代表公司参加了全球微服务架构高峰论坛。会上，Josh Long 对 Spring Cloud 的讲解在现场引起了不小的轰动，也让笔者眼前一亮。会后笔者咨询 Josh，Spring Cloud 能否用于生产、是否大规模使用、国内是否已有成功案例，对方一一给出了肯定的回答。这一回答消除了笔者最后的一点疑虑，开始考虑从 Dubbo 逐步迁移至

Spring Cloud 的规划与方案。会后，笔者心想，不妨将 Spring Cloud 相关知识总结成一个“系列博客”，一来是加深自己的理解，二来也算是丰富 Spring Cloud 的中文资料。于是，笔者创建了自己的博客（*http://www.itmuch.com*），并开始了系列博客的编写。写了两篇后，笔者将博客链接分享到微信群中，没成想，恰好被 Josh Long 看到，并引用到 Spring 官方博客中去了。这让笔者感到无比荣幸的同时，也让自己贡献开源社区的欲望空前强烈，于是乎，一口气又写了两篇。

再后来，笔者成立了微服务/Spring Cloud/Docker 相关的 QQ 群（157525002），在 QQ 群小伙伴的鼓励下，笔者决定写一本 Spring Cloud 开源书（*https://github.com/eacdy/spring-cloud-book*），没想到竟然获得开源中国的推荐。再然后，笔者在许进的邀请下，联合创办了 Spring Cloud 中国社区。最后，在群管理员冯靖的引荐下，认识了网红级的大牛张开涛，开涛帮忙引荐了电子工业出版社编辑侠少。从此，笔者正式撰写实体书。

本以为，有了开源书的撰写经验，实体书应该是较为轻松的一件事。然而，样稿发出后，却被侠少鄙视……主要是语文是体育老师教的，病句满天飞，况且，理论不是我的专长。期间一度想要放弃，多亏了侠少的鼓励，总算坚持写了下去……

仓促完稿之际，感慨万千，激动与感激交织，于是，本段不可免俗，进入老生常谈的“鸣谢”环节——感谢我的家人，写书是个费时费力的活儿，在近半年的时间，我的父母和妻子给予了我极大的支持；感谢电子工业出版社小伙伴们的辛苦工作，没有刘佳禾、孙奇俏、侠少等可爱的朋友们，我的书不可能面世；衷心感谢丁露、冯靖、张英磊、王天青、吴峻申（N 本书的作者）在百忙之中帮忙校对；衷心感谢 QQ 群、微信群的朋友们，你们给了笔者最大的帮助和支持！（注：排名不分先后。）

特别鸣谢：感谢吴峻申给笔者提出很多中肯实用的建议和意见；感谢张英磊帮忙重绘、美化书中绝大部分架构图。

谨以此书献给想要学习微服务、Spring Cloud、Docker 又不知从何开始的读者朋友们。**希望本书能切切实实地帮助你使用特定技术栈实现微服务架构的落地**，也希望本书不会令你失望。本书很多理论性的内容并未展开，例如 Cloud Native、12-factor App、DDD 等，但笔者都在文中以 TIPS、拓展阅读或 WARNING 的形式进行了标记，这部分内容希望读者能够自行拓展阅读。

周立　千米网

排版约定

在阅读本书时，你会遇到不同类型的提示信息，这里展示一些例子及其含义。

推荐阅读按此格式展示。

Tips 按此格式展示。

Warning 按此格式展示。

> **测试**
>
> 测试按此格式展示。

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），您即可享受以下服务。

- **下载资源**：本书所提供的示例代码及资源文件均可在【下载资源】处下载。
- **提交勘误**：您对书中内容的修改意见可在【提交勘误】处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **与作者交流**：在页面下方【读者评论】处留下您的疑问或观点，与作者和其他读者一同学习交流。

页面入口：*http://www.broadview.com.cn/31271*

二维码：

配套代码下载地址

- 1-11 章配套代码：

  https://github.com/itmuch/spring-cloud-docker-microservice-book-code

- 12-14 章配套代码：

  https://github.com/itmuch/spring-cloud-docker-microservice-book-code-docker

目录

1 微服务架构概述 1
1.1 单体应用架构存在的问题 1
1.2 如何解决单体应用架构存在的问题 3
1.3 什么是微服务 3
1.4 微服务架构的优点与挑战 4
1.4.1 微服务架构的优点 5
1.4.2 微服务架构面临的挑战 5
1.5 微服务设计原则 6
1.6 如何实现微服务架构 7
1.6.1 技术选型 7
1.6.2 架构图及常用组件 8

2 微服务开发框架——Spring Cloud 9
2.1 Spring Cloud 简介 9
2.2 Spring Cloud 特点 10
2.3 Spring Cloud 版本 10
2.3.1 版本简介 10
2.3.2 子项目一览 11
2.3.3 Spring Cloud/Spring Boot 版本兼容性 12

3 开始使用 Spring Cloud 实战微服务 13
3.1 Spring Cloud 实战前提 13
3.1.1 技术储备 13
3.1.2 工具及软件版本 14
3.2 服务提供者与服务消费者 15

3.3 编写服务提供者 15
3.3.1 手动编写项目 15
3.3.2 使用 Spring Initializr 快速创建 Spring Boot 项目 20
3.4 编写服务消费者 22
3.5 为项目整合 Spring Boot Actuator 23
3.6 硬编码有哪些问题 26

4 微服务注册与发现 27
4.1 服务发现简介 27
4.2 Eureka 简介 29
4.3 Eureka 原理 29
4.4 编写 Eureka Server 31
4.5 将微服务注册到 Eureka Server 上 33
4.6 Eureka Server 的高可用 34
4.7 为 Eureka Server 添加用户认证 37
4.8 Eureka 的元数据 39
4.8.1 改造用户微服务 39
4.8.2 改造电影微服务 39
4.9 Eureka Server 的 REST 端点 41
4.10 Eureka 的自我保护模式 49
4.11 多网卡环境下的 IP 选择 50
4.12 Eureka 的健康检查 51

5 使用 Ribbon 实现客户端侧负载均衡 53
5.1 Ribbon 简介 53
5.2 为服务消费者整合 Ribbon 54
5.3 使用 Java 代码自定义 Ribbon 配置 57
5.4 使用属性自定义 Ribbon 配置 60
5.5 脱离 Eureka 使用 Ribbon 61

6 使用 Feign 实现声明式 REST 调用 63
6.1 Feign 简介 64

6.2 为服务消费者整合 Feign 64
6.3 自定义 Feign 配置 66
6.4 手动创建 Feign 69
6.4.1 修改用户微服务 70
6.4.2 修改电影微服务 73
6.5 Feign 对继承的支持 75
6.6 Feign 对压缩的支持 76
6.7 Feign 的日志 77
6.8 使用 Feign 构造多参数请求 79
6.8.1 GET 请求多参数的 URL 79
6.8.2 POST 请求包含多个参数 81

7 使用 Hystrix 实现微服务的容错处理 82

7.1 实现容错的手段 82
7.1.1 雪崩效应 83
7.1.2 如何容错 83
7.2 使用 Hystrix 实现容错 85
7.2.1 Hystrix 简介 85
7.2.2 通用方式整合 Hystrix 86
7.2.3 Hystrix 断路器的状态监控与深入理解 89
7.2.4 Hystrix 线程隔离策略与传播上下文 90
7.2.5 Feign 使用 Hystrix 93
7.3 Hystrix 的监控 98
7.4 使用 Hystrix Dashboard 可视化监控数据 100
7.5 使用 Turbine 聚合监控数据 102
7.5.1 Turbine 简介 102
7.5.2 使用 Turbine 监控多个微服务 103
7.5.3 使用消息中间件收集数据 105

8 使用 Zuul 构建微服务网关 110

8.1 为什么要使用微服务网关 110
8.2 Zuul 简介 112

8.3 编写 Zuul 微服务网关.. 112
8.4 Zuul 的路由端点.. 115
8.5 路由配置详解.. 116
8.6 Zuul 的安全与 Header.. 119
8.6.1 敏感 Header 的设置.. 119
8.6.2 忽略 Header.. 120
8.7 使用 Zuul 上传文件.. 121
8.8 Zuul 的过滤器.. 124
8.8.1 过滤器类型与请求生命周期.. 124
8.8.2 编写 Zuul 过滤器.. 125
8.8.3 禁用 Zuul 过滤器.. 127
8.9 Zuul 的容错与回退.. 127
8.10 Zuul 的高可用.. 130
8.10.1 Zuul 客户端也注册到了 Eureka Server 上.. 130
8.10.2 Zuul 客户端未注册到 Eureka Server 上.. 131
8.11 使用 Sidecar 整合非 JVM 微服务.. 132
8.11.1 编写 Node.js 微服务.. 133
8.11.2 编写 Sidecar.. 134
8.11.3 Sidecar 的端点.. 136
8.11.4 Sidecar 与 Node.js 微服务分离部署.. 136
8.11.5 Sidecar 原理分析.. 137
8.12 使用 Zuul 聚合微服务.. 139

9 使用 Spring Cloud Config 统一管理微服务配置.. 144
9.1 为什么要统一管理微服务配置.. 144
9.2 Spring Cloud Config 简介.. 145
9.3 编写 Config Server.. 146
9.4 编写 Config Client.. 149
9.5 Config Server 的 Git 仓库配置详解.. 151
9.6 Config Server 的健康状况指示器.. 154
9.7 配置内容的加解密.. 155
9.7.1 安装 JCE.. 155

9.7.2 Config Server 的加解密端点 155
9.7.3 对称加密 155
9.7.4 存储加密的内容 156
9.7.5 非对称加密 157
9.8 使用/refresh 端点手动刷新配置 158
9.9 使用 Spring Cloud Bus 自动刷新配置 159
9.9.1 Spring Cloud Bus 简介 159
9.9.2 实现自动刷新 160
9.9.3 局部刷新 161
9.9.4 架构改进 162
9.9.5 跟踪总线事件 163
9.10 Spring Cloud Config 与 Eureka 配合使用 163
9.11 Spring Cloud Config 的用户认证 164
9.12 Config Server 的高可用 166
9.12.1 Git 仓库的高可用 166
9.12.2 RabbitMQ 的高可用 167
9.12.3 Config Server 自身的高可用 167

10 使用 Spring Cloud Sleuth 实现微服务跟踪 169

10.1 为什么要实现微服务跟踪 169
10.2 Spring Cloud Sleuth 简介 170
10.3 整合 Spring Cloud Sleuth 171
10.4 Spring Cloud Sleuth 与 ELK 配合使用 174
10.5 Spring Cloud Sleuth 与 Zipkin 配合使用 178
10.5.1 Zipkin 简介 178
10.5.2 编写 Zipkin Server 178
10.5.3 微服务整合 Zipkin 179
10.5.4 使用消息中间件收集数据 183
10.5.5 存储跟踪数据 185

11 Spring Cloud 常见问题与总结 188
11.1 Eureka 常见问题 188
11.1.1 Eureka 注册服务慢 188
11.1.2 已停止的微服务节点注销慢或不注销 189
11.1.3 如何自定义微服务的 Instance ID 190
11.1.4 Eureka 的 UNKNOWN 问题总结与解决 192
11.2 Hystrix/Feign 整合 Hystrix 后首次请求失败 193
11.2.1 原因分析 193
11.2.2 解决方案 193
11.3 Turbine 聚合的数据不完整 193
11.4 Spring Cloud 各组件配置属性 195
11.4.1 Spring Cloud 的配置 195
11.4.2 原生配置 196
11.5 Spring Cloud 定位问题思路总结 196

12 Docker 入门 199
12.1 Docker 简介 199
12.2 Docker 的架构 199
12.3 安装 Docker 201
12.3.1 系统要求 201
12.3.2 移除非官方软件包 201
12.3.3 设置 Yum 源 201
12.3.4 安装 Dokcer 202
12.3.5 卸载 Docker 203
12.4 配置镜像加速器 204
12.5 Docker 常用命令 204
12.5.1 Docker 镜像常用命令 205
12.5.2 Docker 容器常用命令 206

13 将微服务运行在 Docker 上 211
13.1 使用 Dockerfile 构建 Docker 镜像 211
13.1.1 Dockerfile 常用指令 212

13.1.2 使用 Dockerfile 构建镜像 216
13.2 使用 Docker Registry 管理 Docker 镜像 218
13.2.1 使用 Docker Hub 管理镜像 218
13.2.2 使用私有仓库管理镜像 220
13.3 使用 Maven 插件构建 Docker 镜像 222
13.3.1 快速入门 222
13.3.2 插件读取 Dockerfile 进行构建 224
13.3.3 将插件绑定在某个 phase 执行 225
13.3.4 推送镜像 226
13.4 常见问题与总结 228

14 使用 Docker Compose 编排微服务 229

14.1 Docker Compose 简介 229
14.2 安装 Docker Compose 230
14.2.1 安装 Compose 230
14.2.2 安装 Compose 命令补全工具 230
14.3 Docker Compose 快速入门 231
14.3.1 基本步骤 231
14.3.2 入门示例 231
14.3.3 工程、服务、容器 232
14.4 docker-compose.yml 常用命令 232
14.5 docker-compose 常用命令 236
14.6 Docker Compose 网络设置 238
14.6.1 基本概念 238
14.6.2 更新容器 239
14.6.3 links 239
14.6.4 指定自定义网络 239
14.6.5 配置默认网络 240
14.6.6 使用已存在的网络 241
14.7 综合实战：使用 Docker Comose 编排 Spring Cloud 微服务 241
14.7.1 编排 Spring Cloud 微服务 241
14.7.2 编排高可用的 Eureka Server 245

14.7.3 编排高可用 Spring Cloud 微服务集群及动态伸缩 246
14.8 常见问题与总结 249

后记 **250**

1 微服务架构概述

微服务架构是当前软件开发领域的技术热点。它在各种博客、社交媒体和会议演讲上的出镜率非常之高，笔者相信大家也都听说过微服务这个名词。然而微服务似乎又是非常虚幻的——我们找不到微服务的完整定义，以至于很多人认为是在炒作概念。

那么什么是微服务呢？它解决了哪些问题？它又具有哪些特点？诸多问题，本章都将为你一一解答。同时，微服务理论性的内容，互联网上已有很多，本书不会过多提及。笔者会尽量把篇幅花在微服务的具体实战内容上。

1.1 单体应用架构存在的问题

一个归档包（例如 war 格式）包含所有功能的应用程序，通常称为单体应用。而架构单体应用的方法论，就是单体应用架构。

以一个电影售票系统为例，架构如图 1-1 所示。

如图 1-1 所示，该应用尽管已经进行了模块化，但由于 UI 和若干业务模块最终都被打包在一个 war 包中，该 war 包包含了整个系统所有的业务功能，这样的应用系统称为单体应用。

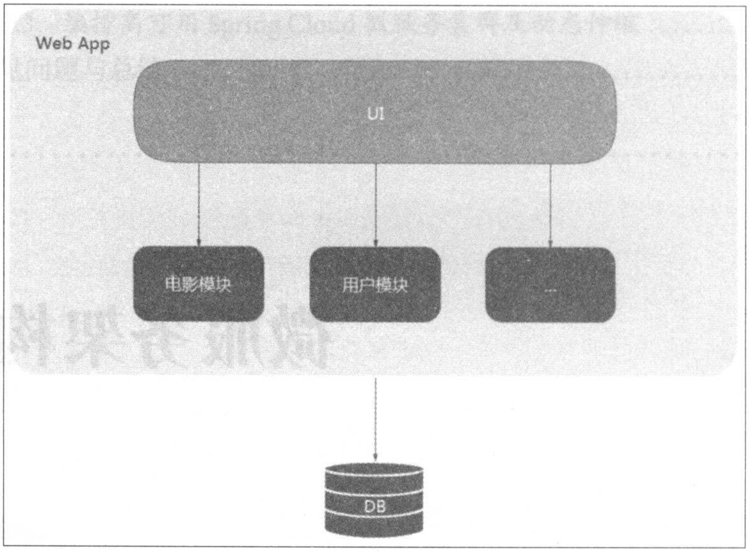


图 1-1 电影售票系统单体架构示意图

相信很多项目都是从单体应用开始的。单体应用比较容易部署、测试，在项目的初期，单体应用可以很好地运行。然而，随着需求的不断增加，越来越多的人加入开发团队，代码库也在飞速地膨胀。慢慢地，单体应用变得越来越臃肿，可维护性、灵活性逐渐降低，维护成本越来越高。下面列举了单体应用存在的一些问题：

- 复杂性高：以笔者经手的一个百万行级别的单体应用为例，整个项目包含的模块非常多、模块的边界模糊、依赖关系不清晰、代码质量参差不齐、混乱地堆砌在一起……整个项目非常复杂。每次修改代码都心惊胆战，甚至添加一个简单的功能，或者修改一个 Bug 都会带来隐含的缺陷。
- 技术债务：随着时间推移、需求变更和人员更迭，会逐渐形成应用程序的技术债务，并且越积越多。"不坏不修（Not broken, don't fix）"，这在软件开发中非常常见，在单体应用中这种思想更甚。已使用的系统设计或代码难以被修改，因为应用程序中的其他模块可能会以意料之外的方式使用它。
- 部署频率低：随着代码的增多，构建和部署的时间也会增加。而在单体应用中，每次功能的变更或缺陷的修复都会导致需要重新部署整个应用。全量部署的方式耗时长、影响范围大、风险高，这使得单体应用项目上线部署的频率较低。而部署频率低又导致两次发布之间会有大量的功能变更和缺陷修复，出错概率比较高。
- 可靠性差：某个应用 Bug，例如死循环、OOM 等，可能会导致整个应用的崩溃。

- 扩展能力受限：单体应用只能作为一个整体进行扩展，无法根据业务模块的需要进行伸缩。例如，应用中有的模块是计算密集型的，它需要强劲的 CPU；有的模块则是 IO 密集型的，需要更大的内存。由于这些模块部署在一起，不得不在硬件的选择上做出妥协。
- 阻碍技术创新：单体应用往往使用统一的技术平台或方案解决所有的问题，团队中的每个成员都必须使用相同的开发语言和框架，要想引入新框架或新技术平台会非常困难。例如，一个使用 Struts 2 构建的、有 100 万行代码的单体应用，如果想要换用 Spring MVC，毫无疑问切换的成本是非常高的。

综上，随着业务需求的发展，功能的不断增加，单体架构很难满足互联网时代业务快速变化的需要。那么，如何解决单体应用架构存在的问题呢？

1.2 如何解决单体应用架构存在的问题

综上所述，单体应用架构存在很多的问题。有没有一种架构模式可以有助于解决这些问题呢？

微服务就是这样的一种架构模式。下面将着重介绍什么是微服务，以及使用微服务架构有哪些优点与挑战。

1.3 什么是微服务

就目前来看，微服务本身并没有一个严格的定义，每个人对微服务的理解都不同。Martin Fowler 在他的博客中是这样描述微服务的。

> In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

用中文表述就是，微服务架构风格是一种将一个单一应用程序开发为一组小型服务的方法，每个服务运行在自己的进程中，服务间通信采用轻量级通信机制（通常用 HTTP 资源 API）。这些服务围绕业务能力构建并且可通过全自动部署机制独立部署。这些服务共用一个最小型的集中式的管理，服务可用不同的语言开发，使用不同的数据存储技术。

从中可以看到，微服务架构应具备以下特性：

- 每个微服务可独立运行在自己的进程里。
- 一系列独立运行的微服务共同构建起整个系统。
- 每个服务为独立的业务开发，一个微服务只关注某个特定的功能，例如订单管理、用户管理等。
- 微服务之间通过一些轻量的通信机制进行通信，例如通过 RESTful API 进行调用。
- 可以使用不同的语言与数据存储技术。
- 全自动的部署机制。

还以电影售票系统为例，使用微服务来架构该应用，架构图如图 1-2 所示。

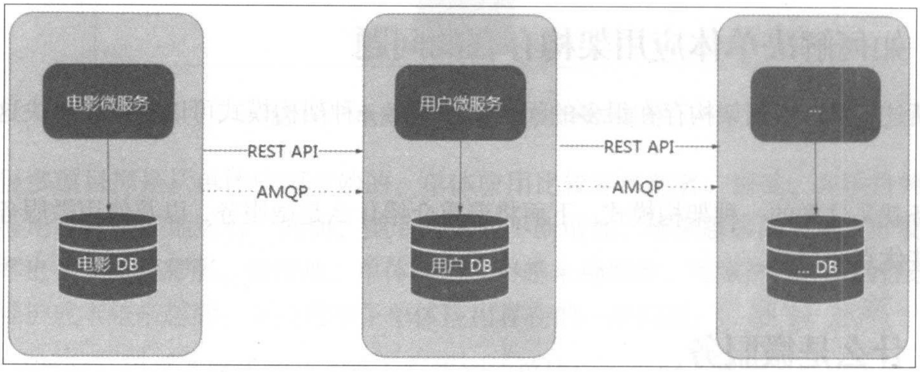


图 1-2 电影售票系统微服务架构示意图

将整个应用分解为多个微服务，各个微服务独立运行在自己的进程中，并分别有自己的数据库，微服务之间使用 REST 或者其他协议通信。

> Martin Fowler《微服务》博客原文：*http://www.martinfowler.com/articles/microservices.html*，译文：*http://blog.cuicc.com/blog/2015/07/22/microservices/*。

1.4 微服务架构的优点与挑战

相对单体应用架构来说，微服务架构有着显著的优点。但是，微服务并非是完美的，使用微服务也为我们的工作带来了一定的挑战。先来分析一下使用微服务有哪些优点。

1.4.1 微服务架构的优点

微服务架构有如下优点。

- 易于开发和维护：一个微服务只会关注一个特定的业务功能，所以它业务清晰、代码量较少。开发和维护单个微服务相对简单。而整个应用是由若干个微服务构建而成的，所以整个应用也会被维持在一个可控状态。
- 单个微服务启动较快：单个微服务代码量较少，所以启动会比较快。
- 局部修改容易部署：单体应用只要有修改，就得重新部署整个应用，微服务解决了这样的问题。一般来说，对某个微服务进行修改，只需要重新部署这个服务即可。
- 技术栈不受限：在微服务架构中，可以结合项目业务及团队的特点，合理地选择技术栈。例如某些服务可使用关系型数据库 MySQL；某些微服务有图形计算的需求，可以使用 Neo4j；甚至可根据需要，部分微服务使用 Java 开发，部分微服务使用 Node.js 开发。
- 按需伸缩：可根据需求，实现细粒度的扩展。例如，系统中的某个微服务遇到了瓶颈，可以结合这个微服务的业务特点，增加内存、升级 CPU 或者是增加节点。

综上所述，单体应用架构的缺点，恰恰是微服务的优点，而这些优点使得微服务看起来简直非常完美。然而完美的东西并不存在，就像银弹不存在一样。下面来讨论使用微服务会带来哪些挑战。

1.4.2 微服务架构面临的挑战

微服务虽然有很多吸引人的地方，但它并不是免费的午餐，使用它是有代价的。本节将讨论使用微服务架构面临的挑战。

- 运维要求较高：更多的服务意味着更多的运维投入。在单体架构中，只需要保证一个应用的正常运行。而在微服务中，需要保证几十甚至几百个服务的正常运行与协作，这给运维带来了很大的挑战。
- 分布式固有的复杂性：使用微服务构建的是分布式系统。对于一个分布式系统，系统容错、网络延迟、分布式事务等都会带来巨大的挑战。
- 接口调整成本高：微服务之间通过接口进行通信。如果修改某一个微服务的 API，可能所有使用了该接口的微服务都需要做调整。
- 重复劳动：很多服务可能都会使用到相同的功能，而这个功能并没有达到分解为一个微服务的程度，这个时候，可能各个服务都会开发这一功能，从而导致代码重复。

尽管可以使用共享库来解决这个问题（例如可以将这个功能封装成公共组件，需要该功能的微服务引用该组件），但共享库在多语言环境下就不一定行得通了。

1.5 微服务设计原则

和数据库设计中的 N 范式一样，微服务也有一定的设计原则，这些原则指导我们更加合理地架构微服务。

- 单一职责原则

  单一职责原则指的是一个单元（类、方法或者服务等）只应关注整个系统功能中单独、有界限的一部分。单一职责原则可以帮助我们更优雅地开发、更敏捷地交付。

  单一职责原则是 SOLID 原则之一。有兴趣的读者可前往*https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)* 进行扩展阅读。

- 服务自治原则

  服务自治是指每个微服务应具备独立的业务能力、依赖与运行环境。在微服务架构中，服务是独立的业务单元，应该与其他服务高度解耦。每个微服务从开发、测试、构建、部署，都应当可以独立运行，而不应该依赖其他的服务。

- 轻量级通信机制

  微服务之间应该通过轻量级的通信机制进行交互。笔者认为，轻量级的通信机制应具备两点：首先是它的体量较轻；其次是它应该是跨语言、跨平台的。例如我们所熟悉的 REST 协议，就是一个典型的“轻量级通信机制”；而例如 Java 的 RMI 则协议就不大符合轻量级通信机制的要求，因为它绑定了 Java 语言。

  微服务架构中，常用的协议有 REST、AMQP、STOMP、MQTT 等。

- 微服务粒度

  微服务的粒度是难点，也常常是争论的焦点。应当使用合理的粒度划分微服务，而不是一味地把服务做小。代码量的多少不能作为微服务划分的依据，因为不同的微服务本身的业务复杂性不同，代码量也不同。

  在微服务的设计阶段，就应确定其边界。微服务之间应相对独立并保持松耦合。笔者认为，领域驱动设计（Domain Driven Design，简称 DDD）中的“界限上下文（Bounded Context）”可作为划分微服务边界、确定微服务粒度的重要依据。限于篇幅，DDD 的内容无法展开讲解，对 DDD 感兴趣的读者朋友们可阅读《Domain Driven Design Quickly》快速入门，也可阅读 DDD 开山鼻祖 Eric Evans 的《领域驱动设计：软件核心复杂性应对之道》深入理解。同时，在划分微服务的过程中，还应综合考量团队

的现状。康威定律（Conway's Law）相信大家已经很熟悉了，笔者不再赘述。

微服务架构的演进是一个循序渐进的过程。在演进过程中，常常会根据业务的变化，对微服务进行重构，甚至是重新划分，从而让架构更加合理。最终，当微服务的开发、部署、测试以及运维的效率很高，并且成本很低时，一个好的微服务架构就形成了。

- 《Domain Driven Design Quickly》阅读地址：*https://www.infoq.com/minibooks/domain-driven-design-quickly/*。
- 《Domain Driven Design Quickly》中文版《领域驱动设计精简版》：*http://www.infoq.com/cn/minibooks/domain-driven-design-quickly-new*。

1.6 如何实现微服务架构

至此，不仅知道了微服务架构的定义及其优缺点，还总结了一些指导性的原则，为合理架构微服务提供了理论支持。

下面来探讨一下，如何实现微服务架构。

1.6.1 技术选型

相对单体应用的交付，微服务应用的交付要复杂很多，不仅需要开发框架的支持，还需要一些自动化的部署工具，以及 IaaS、PaaS 或 CaaS 的支持。

下面从开发和运行平台两个维度考虑挑选技术选型。

- 开发框架的选择

  可使用 Spring Cloud 作为微服务开发框架。

  首先，Spring Cloud 具备开箱即用的生产特性，可大大提升开发效率；再者，Spring Cloud 的文档丰富、社区活跃，遇到问题比较容易获得支持；更为可贵的是，Spring Cloud 为微服务架构提供了完整的解决方案。

  当然，也可使用其他的开发框架或者解决方案来实现微服务，例如 Dubbo、Dropwizard、Armada 等。

- 运行平台

  微服务并不绑定运行平台，将微服务部署在 PC Server，或者阿里云、AWS 等云计算平台都是可以的。出于轻量、灵活、应用支撑等方面的考虑，本书将演示如何在 Docker 上部署微服务。

- Dubbo 的 GitHub：*https://github.com/alibaba/dubbo*。
- Dropwizard 的 GitHub：*https://github.com/dropwizard/dropwizard*。
- Armada 的 GitHub：*https://github.com/armadaplatform/armada*。

1.6.2 架构图及常用组件

在进入实战之前，先来通览一下微服务架构，如图 1-3 所示。

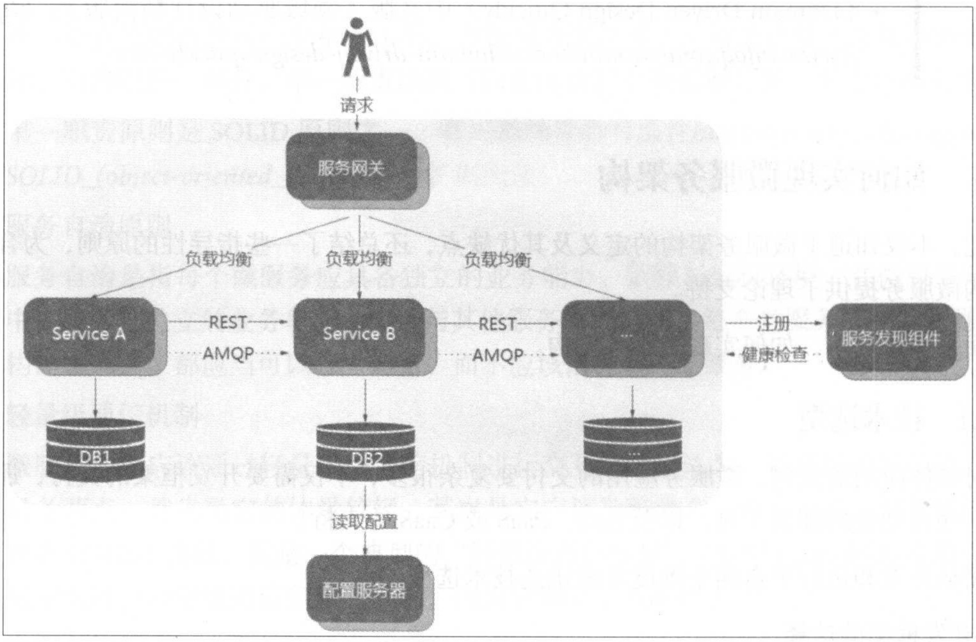


图 1-3 微服务架构图

图 1-3 不严谨地表示了一个微服务应用的架构。图中将所有服务都注册到服务发现组件上，服务之间使用轻量级的通信机制通信。由图可以看到，除了 service A、service B 等，还有服务发现组件、服务网关、配置服务器等组件。这些组件分别是什么？它们的作用又是什么？

读者可以带着疑问继续阅读下去，本书将在实战的过程中进行详细的讲解。

之所以说图 1-3 不严谨，是因为配置服务器可以注册到服务发现组件上；而服务发现组件也可以从配置服务器读取配置信息。

2 微服务开发框架——Spring Cloud

2.1 Spring Cloud 简介

尽管 Spring Cloud 带有“Cloud”的字样，但它并不是云计算解决方案，而是在 Spring Boot 基础上构建的，用于快速构建分布式系统的通用模式的工具集。

使用 Spring Cloud 开发的应用程序非常适合在 Docker 或者 PaaS（例如 Cloud Foundry）上部署，所以又叫作云原生应用（Cloud Native Application）。云原生（Cloud Native）可简单理解为面向云环境的软件架构。说到云原生，就不得不提一下《十二要素应用宣言（12-factor Apps）》，这是云原生架构的方法论与最佳实践。限于篇幅，本书不作赘述，有兴趣的读者可参考本节的拓展阅读。

- 《Cloud Native Application》电子书：*https://pivotal.io/platform-as-a-service/migrating-to-cloud-native-application-architectures-ebook*。
- 《十二要素应用宣言（12-factor Apps）》中文版：*https://12factor.net/zh_cn/*。

2.2 Spring Cloud 特点

Spring Cloud 有以下特点：

- 约定优于配置。
- 适用于各种环境。开发、部署在 PC Server 或各种云环境（例如阿里云、AWS 等）均可。
- 隐藏了组件的复杂性，并提供声明式、无 xml 的配置方式。
- 开箱即用，快速启动。
- 轻量级的组件。Spring Cloud 整合的组件大多比较轻量。例如 Eureka、Zuul，等等，都是各自领域轻量级的实现。
- 组件丰富，功能齐全。Spring Cloud 为微服务架构提供了非常完整的支持。例如，配置管理、服务发现、断路器、微服务网关等。
- 选型中立、丰富。例如，Spring Cloud 支持使用 Eureka、Zookeeper 或 Consul 实现服务发现。
- 灵活。Spring Cloud 的组成部分是解耦的，开发人员可按需灵活挑选技术选型。

2.3 Spring Cloud 版本

大多数 Spring 项目都是以“主版本号.次版本号.增量版本号.里程碑版本号”的形式命名版本号的，例如 Spring Framework 稳定版本 4.3.5.RELEASE、里程碑版本 5.0.0.M4 等。其中，主版本号表示项目的重大重构；次版本号表示新特性的添加和变化；增量版本号一般表示 Bug 修复；里程碑版本号表示某版本号的里程碑。

然而，Spring Cloud 并未使用这种方式管理版本。下面来详细探讨一下 Spring Cloud 的版本。

2.3.1 版本简介

先看一下 Spring Cloud 的版本，如图 2-1 所示。

由图 2-1 可知，Spring Cloud 是以英文单词 SRX（X 为数字）的形式命名版本号的。那么英文单词和 SR 分别表示什么呢？

Spring Cloud 是一个综合项目，它包含很多的子项目。由于子项目也维护着自己的版本号，Spring Cloud 采用了这种版本命名方式，从而避免与子项目的版本混淆。其中，英

文单词叫作“release train”，Angel、Brixton、Camden 等都是伦敦地铁站的名称，它们按照字母顺序发行，可将其理解为主版本的演进。SR 表示“Service Release”，一般表示 Bug 修复；在 SR 版本发布之前，会先发布一个 Release 版本，例如 Camden RELEASE。

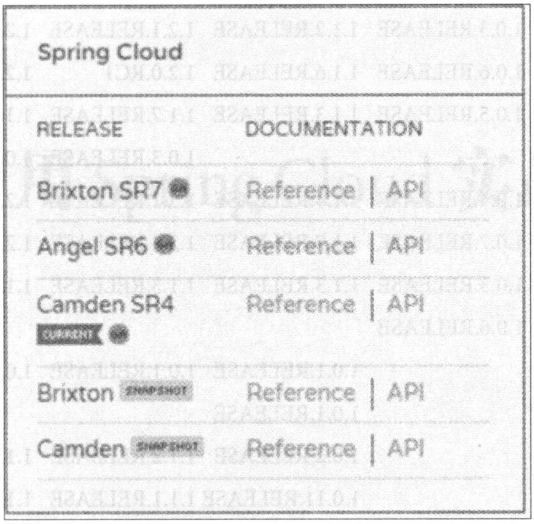


图 2-1 Spring Cloud 版本

经过以上讲解，相信大家就能很好地理解 Spring Cloud 的版本了。例如，Camden SR3 表示 Camden 版本的第 3 次 Bug 修复版本。

- Spring Cloud 版本发布记录可详见：*https://github.com/spring-cloud/spring-cloud-release/releases*，从中可清晰地看到 Spring Cloud 版本发布的时间及先后顺序。
- Spring Cloud 版本演进计划：*https://github.com/spring-cloud/spring-cloud-release/milestones*，从中可了解 Spring Cloud 未来的版本演进计划。
- 事实上，Spring 有不少项目使用类似的命名方式。例如 Spring Data（*http://projects.spring.io/spring-data/*）、Spring Cloud Stream（*http://cloud.spring.io/spring-cloud-stream/*）等。

2.3.2 子项目一览

理解 Spring Cloud 的版本后，来看一下各版本 Spring Cloud 包含的子项目及版本，如表 2-1 所示。

表 2-1 Spring Cloud 各版本组件

Component	Angel.SR6	Brixton.SR7	Camden.SR4	Camden.BUILD-SNAPSHOT
spring-cloud-aws	1.0.4.RELEASE	1.1.3.RELEASE	1.1.3.RELEASE	1.1.4.BUILD-SNAPSHOT
spring-cloud-bus	1.0.3.RELEASE	1.1.2.RELEASE	1.2.1.RELEASE	1.2.2.BUILD-SNAPSHOT
spring-cloud-cli	1.0.6.RELEASE	1.1.6.RELEASE	1.2.0.RC1	1.2.0.BUILD-SNAPSHOT
spring-cloud-commons	1.0.5.RELEASE	1.1.3.RELEASE	1.1.7.RELEASE	1.1.8.BUILD-SNAPSHOT
spring-cloud-contract			1.0.3.RELEASE	1.0.4.BUILD-SNAPSHOT
spring-cloud-config	1.0.4.RELEASE	1.1.3.RELEASE	1.2.1.RELEASE	1.2.2.BUILD-SNAPSHOT
spring-cloud-netflix	1.0.7.RELEASE	1.1.7.RELEASE	1.2.4.RELEASE	1.2.5.BUILD-SNAPSHOT
spring-cloud-security	1.0.3.RELEASE	1.1.3.RELEASE	1.1.3.RELEASE	1.1.3.BUILD-SNAPSHOT
spring-cloud-starters	1.0.6.RELEASE			
spring-cloud-cloudfoundry		1.0.1.RELEASE	1.0.1.RELEASE	1.0.2.BUILD-SNAPSHOT
spring-cloud-cluster		1.0.1.RELEASE		
spring-cloud-consul		1.0.2.RELEASE	1.1.2.RELEASE	1.1.3.BUILD-SNAPSHOT
spring-cloud-sleuth		1.0.11.RELEASE	1.1.1.RELEASE	1.1.2.BUILD-SNAPSHOT
spring-cloud-stream		1.0.2.RELEASE	Brooklyn.RC1	Brooklyn.BUILD-SNAPSHOT
spring-cloud-zookeeper		1.0.3.RELEASE	1.0.3.RELEASE	1.0.4.BUILD-SNAPSHOT
spring-boot	1.2.8.RELEASE	1.3.8.RELEASE	1.4.2.RELEASE	1.4.2.BUILD-SNAPSHOT
spring-cloud-task		1.0.3.RELEASE	1.0.3.RELEASE	1.0.4.BUILD-SNAPSHOT

从中不难发现，Angel 版本包含的子项目相对较少；Brixton、Camden 则提供了更多的组件。相信随着项目的迭代，Spring Cloud 会提供更多的组件与更丰富的特性，从而让开发更加简单、快速。

2.3.3 Spring Cloud/Spring Boot 版本兼容性

- Angel 版本基于 Spring Boot 1.2.x 构建，在一些场景下，与 Spring Boot 1.3.x 及以上版本不兼容。
- Brixton 版本基于 Spring Boot 1.3.x 构建，也可使用 1.4.x 进行测试，与 Spring Boot 1.2.x 不兼容。
- Camden 版本基于 Spring Boot 1.4.x 构建，也可使用 1.5.x 进行测试。

读者可前往*http://projects.spring.io/spring-cloud/*查看版本兼容性。

3 开始使用 Spring Cloud 实战微服务

本章正式开始使用 Spring Cloud 进行微服务实践。

3.1 Spring Cloud 实战前提

Spring Cloud 不一定适合所有人。先来探讨一下，玩转 Spring Cloud 需要具备什么样的技术能力，以及在实战中会使用到哪些工具。

3.1.1 技术储备

Spring Cloud 并不是面向零基础开发人员的，它有一定的学习曲线。

- 语言基础：Spring Cloud 是一个基于 Java 语言的工具套件，所以学习它需要一定的 Java 基础。当然，Spring Cloud 同样也支持使用 Scala、Groovy 等语言进行开发。本书的示例代码都是使用 Java 编写的。
- Spring Boot：Spring Cloud 是基于 Spring Boot 构建的，因此它延续了 Spring Boot 的契约模式以及开发方式。如果大家对 Spring Boot 不熟悉，建议花一点时间入门。

  当然，本书会尽量照顾到不熟悉 Spring Boot 的读者，由浅入深地进行讲解。

- 项目管理与构建工具：目前业界比较主流的项目管理与构建工具有 Maven 和 Gradle 等，本书采用的是目前相对主流的 Maven。大家也可使用 Gradle 管理与构建项目。并且，Maven 与 Gradle 项目可以互相转换，例如，使用以下命令即可将 Maven 项目转换为 Gradle 项目。

```
gradle init --type pom
```

3.1.2 工具及软件版本

目前 Spring Cloud 正在飞速地发展，是业界有名的"版本帝"。2015 年 7 月，Spring Cloud 才刚刚发布 Angel RELEASE；而 2016 年 5 月，Spring Cloud 就已经发布到 Brixton RELEASE；到 2016 年 9 月，Spring Cloud 又发布了新一代产品 Camden RELEASE。随着版本的演进，Spring Cloud 带来了更丰富的组件、更强大的功能，并解决了很多遗留的 Bug。

新版本未必代表着完美，但老版本往往意味着过时或即将过时。基于这个原则，笔者使用目前最新的 Release 版本进行讲解。涉及到的新特性，笔者会尽量标记并做出讲解。

下面列出了笔者所使用的各个软件及其版本。

- JDK：Spring Cloud 官方建议使用 JDK 1.8。当然，Spring Cloud 也支持通过一定的配置，使用 JDK 1.7 进行开发。笔者使用的是 JDK 1.8。
- Spring Boot：本书使用 Spring Boot 1.4.3.RELEASE。
- Spring Cloud：本书使用 Spring Cloud Camden SR4。
- IDE 的选择：选择一款强大的 IDE 往往能够事半功倍。笔者使用的 IDE 是 Spring 官方提供的 Spring Tool Suite 3.8.3，这是一个基于 Eclipse 的 IDE。当然，使用 IntelliJ IDEA 等 IDE 进行开发也是可以的。
- Maven：笔者使用 Maven 3.3.9 构建项目。和 Spring Boot、Spring Cloud 一样，Maven 3.3.x 默认也是运行在 JDK 1.8 之上的。如果想使用 1.8 以下版本的 JDK，需要做一些额外的配置。

> 目前，Spring Cloud 版本演进速度很快，不同版本之间有一定差异，建议大家在学习时，尽量选用与本书一致的软件版本。要知道，学习是有成本的，这个成本是时间和精力，降低学习成本的重要方法之一就是少踩坑。因此，建议使用与本书相同的版本进行学习，掌握相关知识并具备解决问题的能力后，再按照项目需求挑选适合生产的版本。

3.2 服务提供者与服务消费者

使用微服务构建的是分布式系统，微服务之间通过网络进行通信。我们使用服务提供者与服务消费者来描述微服务之间的调用关系。表 3-1 解释了服务提供者与服务消费者。

表 3-1 服务提供者与服务消费者

名词	定义
服务提供者	服务的被调用方（即：为其他服务提供服务的服务）
服务消费者	服务的调用方（即：依赖其他服务的服务）

我们继续以电影售票系统为例。如图 3-1，用户向电影微服务发起了一个购票的请求。在进行购票的业务操作前，电影微服务需要调用用户微服务的接口，查询当前用户的余额是多少，是不是符合购票标准等。在这种场景下，用户微服务就是一个服务提供者，电影微服务则是一个服务消费者。

图 3-1 服务提供者与服务消费者

围绕该场景，先来编写一个用户微服务，然后编写一个电影微服务。

3.3 编写服务提供者

本节将编写一个服务提供者（用户微服务），该服务可通过主键查询用户信息。为便于测试，使用 Spring Data JPA 作为持久层框架，使用 H2 作为数据库。

3.3.1 手动编写项目

1. 创建一个 Maven 项目，它的 ArtifactId 是`microservice-simple-provider-user`，pom.xml 的内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org
    /2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
    xsd/maven-4.0.0.xsd">
```

```xml
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.itmuch.cloud</groupId>
  <artifactId>microservice-simple-provider-user</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <!-- 引入spring boot的依赖 -->
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.4.3.RELEASE</version>
  </parent>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <java.version>1.8</java.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
      <groupId>com.h2database</groupId>
      <artifactId>h2</artifactId>
    </dependency>
  </dependencies>

  <!-- 引入spring cloud的依赖 -->
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-dependencies</artifactId>
```

```
        <version>Camden.SR4</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>

  <!-- 添加spring-boot的maven插件 -->
  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>
```

其中，`spring-boot-starter-web` 提供了 Spring MVC 的支持；`spring-boot-starter-data-jpa` 提供了 Spring Data JPA 的支持。

2. 准备好建表语句，在项目的 classpath 下建立 schema.sql，并添加如下内容：

```
drop table user if exists;
create table user (id bigint generated by default as identity, username varchar
(40), name varchar(20), age int(3), balance decimal(10,2), primary key (id));
```

3. 准备几条数据，在项目的 classpath 下建立文件 data.sql，并添加如下内容：

```
insert into user (id, username, name, age, balance) values (1, 'account1',
'张三', 20, 100.00);
insert into user (id, username, name, age, balance) values (2, 'account2',
'李四', 28, 180.00);
insert into user (id, username, name, age, balance) values (3, 'account3',
'王五', 32, 280.00);
```

4. 创建用户实体类：

```
@Entity
public class User {
  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Long id;
```

```
  @Column
  private String username;
  @Column
  private String name;
  @Column
  private Integer age;
  @Column
  private BigDecimal balance;
  ...
  // getters and stters
}
```

5. 创建 DAO：

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
}
```

6. 创建 Controller：

```
@RestController
public class UserController {
  @Autowired
  private UserRepository userRepository;

  @GetMapping("/{id}")
  public User findById(@PathVariable Long id) {
    User findOne = this.userRepository.findOne(id);
    return findOne;
  }
}
```

Controller 中用到的`@GetMapping`，是 Spring 4.3 提供的新注解。它是一个组合注解，等价于`@RequestMapping(method = RequestMethod.GET)`，用于简化开发。同理还有 `@PostMapping`、`@PutMapping`、`@DeleteMapping`、`@PatchMapping`等。

7. 编写启动类，在类上使用`@SpringBootApplication` 声明这是一个 Spring Boot 项目。

```
@SpringBootApplication
public class ProviderUserApplication {
  public static void main(String[] args) {
    SpringApplication.run(ProviderUserApplication.class, args);
  }
}
```

@SpringBootApplication是一个组合注解，它整合了@Configuration、@EnableAutoConfiguration 和@ComponentScan注解，并开启了 Spring Boot 程序的组件扫描和自动配置功能。在开发 Spring Boot 程序的过程中，常常会组合使用 @Configuration、@EnableAutoConfiguration 和@ComponentScan等注解，所以 Spring Boot 提供了 @SpringBootApplication，来简化开发。

8. 编写配置文件，命名为 application.yml。

```
server:
  port: 8000
spring:
  jpa:
    generate-ddl: false
    show-sql: true
    hibernate:
      ddl-auto: none
  datasource:                        # 指定数据源
    platform: h2                     # 指定数据源类型
    schema: classpath:schema.sql     # 指定h2数据库的建表脚本
    data: classpath:data.sql         # 指定h2数据库的数据脚本
logging:                             # 配置日志级别，让hibernate打印执行的SQL
  level:
    root: INFO
    org.hibernate: INFO
    org.hibernate.type.descriptor.sql.BasicBinder: TRACE
    org.hibernate.type.descriptor.sql.BasicExtractor: TRACE
```

在传统的 Web 开发中，常使用 properties 格式文件作为配置文件。Spring Boot 以及 Spring Cloud 支持使用 properties 或者 yml 格式的文件作为配置文件。

yml 文件格式是 YAML（Yet Another Markup Language）编写的文件格式，YAML 和 properties 格式的文件可互相转换，例如本节中的 application.yml，就等价于如下的 properties 文件。

```
server.port=8000
spring.jpa.generate-ddl=false
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=none
spring.datasource.platform=h2
spring.datasource.schema=classpath:schema.sql
spring.datasource.data=classpath:data.sql
```

```
logging.level.root=INFO
logging.level.org.hibernate=INFO
logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE
logging.level.org.hibernate.type.descriptor.sql.BasicExtractor=TRACE
```

从中不难看出，YAML 比 properties 结构清晰；可读性、可维护性也更强，并且语法非常简洁。因此，本书使用 YAML 格式作为配置文件。另外，**yml 有严格的缩进，请大家注意**。

> **测试**
>
> 访问：*http://localhost:8000/1*，获得结果：
>
> ```
> {
>    "id": 1,
>    "username": "account1",
>    "name": "张三",
>    "age": 20,
>    "balance": 100
> }
> ```
>
> 说明已可通过 ID 查询用户信息。

3.3.2 使用 Spring Initializr 快速创建 Spring Boot 项目

之前是手动创建项目的。事实上，也可使用 Spring Initializr 快速创建项目。虽然 Spring Initializr 不能生成应用程序的业务代码，但它可生成基本的项目结构。这样就可以把更多精力放在具体的业务代码上，而无须过分关注项目搭建的过程。

Spring Initializr 有以下几种用法：

- 通过网页使用。
- 通过 Spring Tool Suite 使用。
- 通过 IntelliJ IDEA 使用。
- 使用 Spring Boot CLI 使用。

笔者以第一种方式为例进行讲解，其他方式大致类似，请读者自行发掘。

1. 访问：*http://start.spring.io/*，会看到类似图 3-2 的界面。
2. 按照需求，选择项目类型（Maven 或 Gradle）、Spring Boot 的版本，并填写项目元数据以及所需依赖。如点击“Switch to the full version”按钮，还可指定额外的信息，例如 Java 版本、打包方式等。

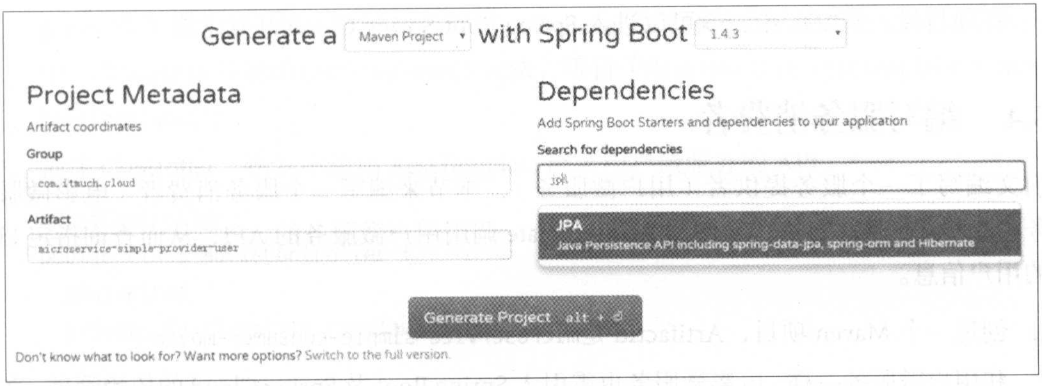


图 3-2 Spring Initializr 首页

3. 点击 Generate Project 按钮，就能获得一个名为 microservice-simple-provider-user.zip 的压缩包文件。解压后，项目结构如下：

```
│  mvnw
│  mvnw.cmd
│  pom.xml
├─.mvn
│  └─wrapper
│          maven-wrapper.jar
│          maven-wrapper.properties
└─src
    ├─main
    │  ├─java
    │  │  └─com
    │  │      └─itmuch
    │  │          └─cloud
    │  │                  MicroserviceSimpleProviderUserApplication.java
    │  └─resources
    │      │  application.properties
    │      ├─static
    │      └─templates
    └─test
        └─java
            └─com
                └─itmuch
                    └─cloud
                            MicroserviceSimpleProviderUserApplicationTests.java
```

将项目导入到 IDE 中，就可以进入 Spring Boot/Spring Cloud 开发之旅了。

3.4　编写服务消费者

前文编写了一个服务提供者（用户微服务），本节来编写一个服务消费者（电影微服务）。该服务非常简单，它使用 RestTemplate 调用用户微服务的 API，从而查询指定 id 的用户信息。

1. 创建一个 Maven 项目，ArtifactId 是microservice-simple-consumer-movie 。
2. 和用户微服务一样，电影微服务也需引入 Spring Boot 及 Spring Cloud 的依赖管理，在此基础上，添加以下依赖。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

3. 创建用户实体类，该类是一个 POJO。

```
public class User {
  private Long id;
  private String username;
  private String name;
  private Integer age;
  private BigDecimal balance;
  ...
  // getters and setters
}
```

4. 创建启动类，代码如下。

```
@SpringBootApplication
public class ConsumerMovieApplication {
  @Bean
  public RestTemplate restTemplate() {
    return new RestTemplate();
  }

  public static void main(String[] args) {
    SpringApplication.run(ConsumerMovieApplication.class, args);
  }
}
```

`@Bean` 是一个方法注解，作用是实例化一个 Bean 并使用该方法的名称命名。在本例中，添加`@Bean` 注解的`restTemplate()` 方法，等价于`RestTemplate restTemplate = new RestTemplate();` 。

5. 创建 Controller，在其中使用 RestTemplate 请求用户微服务的 API。

```
@RestController
public class MovieController {
  @Autowired
  private RestTemplate restTemplate;

  @GetMapping("/user/{id}")
  public User findById(@PathVariable Long id) {
   return this.restTemplate.getForObject("http://localhost:8000/" + id, User.
      class);
  }
}
```

6. 编写配置文件 application.yml：

```
server:
  port: 8010
```

这样，一个电影微服务就完成了，so easy！

测试

访问：*http://127.0.0.1:8010/user/1* ，结果如下：

```
{
   "id": 1,
   "username": "account1",
   "name": "张三",
   "age": 20,
   "balance": 100
}
```

说明电影微服务可以正常使用 RestTemplate 调用用户微服务的 API。

3.5 为项目整合 Spring Boot Actuator

Spring Boot Actuator 提供了很多监控端点。可使用`http://{ip}:{port}/{endpoint}` 的形式访问这些端点，从而了解应用程序的运行状况。

Actuator 提供的端点，如表 3-2 所示。

表 3-2 Spring Boot Actuator 常用端点及描述

端点	描述	HTTP 方法
autoconfig	显示自动配置的信息	GET
beans	显示应用程序上下文所有的 Spring bean	GET
configprops	显示所有 @ConfigurationProperties 的配置属性列表	GET
dump	显示线程活动的快照	GET
env	显示应用的环境变量	GET
health	显示应用程序的健康指标，这些值由 HealthIndicator 的实现类提供	GET
info	显示应用的信息，可使用info.*属性自定义 info 端点公开的数据	GET
mappings	显示所有的 URL 路径	GET
metrics	显示应用的度量标准信息	GET
shutdown	关闭应用（默认情况下不启用，如需启用，需设置 endpoints.shutdown.enabled=true）	POST
trace	显示跟踪信息（默认情况下为最近 100 个 HTTP 请求）	GET

由于在后面有大量的章节需要用到 Actuator，不妨先来为项目整合 Actuator，以项目 microservice-simple-provider-user 为例。

为项目添加以下依赖。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

这样，就整合好 Actuator 了。是不是十分简单呢？同理，也可为项目 microservice-simple-consumer-movie 整合 Actuator。

测试

1. 访问*http://localhost:8000/health*，可获得类似如下的结果。

```
{
    "status": "UP",
    "diskSpace": {
        "status": "UP",
```

```
      "total": 214750457856,
      "free": 183337144320,
      "threshold": 10485760
    },
    "db": {
      "status": "UP",
      "database": "H2",
      "hello": 1
    }
}
```

其中，UP 表示运行正常，除 UP 外，还有 DOWN、OUT_OF_SERVICE、UNKNOWN 等状态。

2. 访问*http://localhost:8000/info*，可看到以下内容。

```
{}
```

由结果可知，info 端点并没有返回任何数据给我们。可使用`info.*`属性来自定义 info 端点公开的数据，例如：

```
info:
  app:
    name: @project.artifactId@
    encoding: @project.build.sourceEncoding@
    java:
      source: @java.version@
      target: @java.version@
```

这样，重启后，再次访问*http://localhost:8000/info*，就会看到类似如下的内容。

```
{
    "app": {
        "name": "microservice-simple-provider-user",
        "encoding": "UTF-8",
        "java": {
            "source": "1.8.0_92",
            "target": "1.8.0_92"
        }
    }
}
```

> 由结果可知，info 端点返回了项目名称、编码、Java 版本等信息。
>
> Actuator 端点众多，其他端点读者可参考 Spring Boot 文档自行测试，笔者不作赘述。

3.6 硬编码有哪些问题

至此，已经实现了一个用户微服务和电影微服务，并在电影微服务中使用 RestTemplate 调用用户微服务中的 RESTful API。一切都是那么的自然、简单、perfect！

那么真的完美吗？来分析一下电影微服务的代码，在 MovieController.java 中：

```
@GetMapping("/user/{id}")
public User findById(@PathVariable Long id) {
  return this.restTemplate.getForObject("http://localhost:8000/" + id, User.class);
}
```

由代码可知，我们是把提供者的网络地址（IP 和端口等）硬编码在代码中的，当然，也可将其提取到配置文件中去。例如：

```
user:
  userServiceUrl: http://localhost:8000/
```

代码改为：

```
@Value("user.userServiceUrl")
private String userServiceUrl;

@GetMapping("/user/{id}")
public User findById(@PathVariable Long id) {
  return this.restTemplate.getForObject(this.userServiceUrl + id, User.class);
}
```

在传统的应用程序中，一般都是这么做的。然而，这种方式有很多问题。

- 适用场景有局限：如果服务提供者的网络地址（IP 和端口）发生了变化，将会影响服务消费者。例如，用户微服务的网络地址发生变化，就需要修改电影微服务的配置，并重新发布。这显然是不可取的。
- 无法动态伸缩：在生产环境中，每个微服务一般都会部署多个实例，从而实现容灾和负载均衡。在微服务架构的系统中，还需要系统具备自动伸缩的能力，例如动态增减节点等。硬编码无法适应这种需求。

那么要如何解决这些问题呢？请大家继续阅读下去。

第4章 微服务注册与发现

4.1 服务发现简介

通过前文的讲解，我们知道硬编码提供者地址的方式有不少问题。要想解决这些问题，服务消费者需要一个强大的服务发现机制，服务消费者使用这种机制获取服务提供者的网络信息。不仅如此，即使服务提供者的信息发生变化，服务消费者也无须修改配置文件。

服务发现组件提供这种能力。在微服务架构中，服务发现组件是一个非常关键的组件。

使用服务发现组件后的架构图，如图 4-1 所示。

服务提供者、服务消费者、服务发现组件这三者之间的关系大致如下：

- 各个微服务在启动时，将自己的网络地址等信息注册到服务发现组件中，服务发现组件会存储这些信息。
- 服务消费者可从服务发现组件查询服务提供者的网络地址，并使用该地址调用服务提供者的接口。
- 各个微服务与服务发现组件使用一定机制（例如心跳）通信。服务发现组件如长时间无法与某微服务实例通信，就会注销该实例。

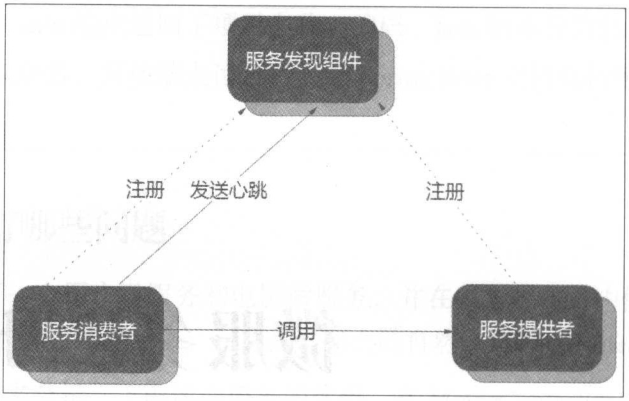


图 4-1 服务发现架构图

- 微服务网络地址发生变更（例如实例增减或者 IP 端口发生变化等）时，会重新注册到服务发现组件。使用这种方式，服务消费者就无须人工修改提供者的网络地址了。

综上，服务发现组件应具备以下功能。

- 服务注册表：是服务发现组件的核心，它用来记录各个微服务的信息，例如微服务的名称、IP、端口等。服务注册表提供查询 API 和管理 API，查询 API 用于查询可用的微服务实例，管理 API 用于服务的注册和注销。
- 服务注册与服务发现：服务注册是指微服务在启动时，将自己的信息注册到服务发现组件上的过程。服务发现是指查询可用微服务列表及其网络地址的机制。
- 服务检查：服务发现组件使用一定机制定时检测已注册的服务，如发现某实例长时间无法访问，就会从服务注册表中移除该实例。

综上，使用服务发现的好处是显而易见的。Spring Cloud 提供了多种服务发现组件的支持，例如 Eureka、Consul 和 Zookeeper 等。本书将以 Eureka 为例，为大家详细讲解服务注册与发现。

- 目前市面上的书籍中所提到的服务注册、服务发现或注册中心等名词，多数场景下都可理解为服务发现组件。
- 服务发现的方式可细分为服务器端发现和客户端发现，由于原理相通，本书不再赘述。

4.2 Eureka 简介

Eureka 是 Netflix 开源的服务发现组件，本身是一个基于 REST 的服务。它包含 Server 和 Client 两部分。Spring Cloud 将它集成在子项目 Spring Cloud Netflix 中，从而实现微服务的注册与发现。

- Eureka 的 GitHub：*https://github.com/Netflix/Eureka*。
- Netflix 是一家在线影片租赁提供商。

4.3 Eureka 原理

在分析 Eureka 的原理之前，先来了解一下 Region 和 Availability Zone，如图 4-2 所示。

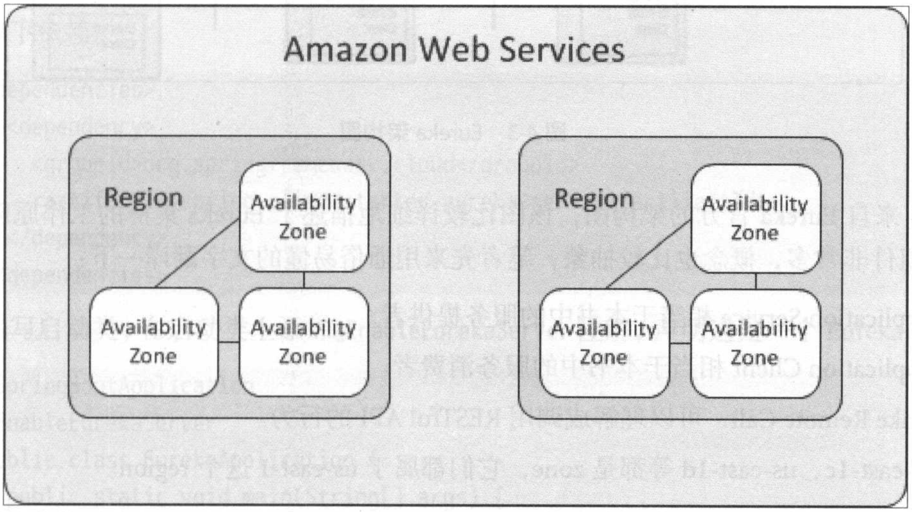


图 4-2 Region 与 Availability Zone

Region 和 Availability Zone 均是 AWS 的概念。其中，Region 表示 AWS 中的地理位置，每个 Region 都有多个 Availability Zone，各个 Region 之间完全隔离。AWS 通过这种方式实现了最大的容错和稳定性。

Spring Cloud 默认使用的 Region 是`us-east-1`，在非 AWS 环境下，可以将 Availability Zone 理解成机房，将 Region 理解为跨机房的 Eureka 集群。

对 Region 和 Availability Zone 感兴趣的读者可前往*http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html* 进行扩展阅读。

理解 Region 和 Availability Zone 后，来分析一下 Eureka 的原理，Eureka 架构如图 4-3所示。

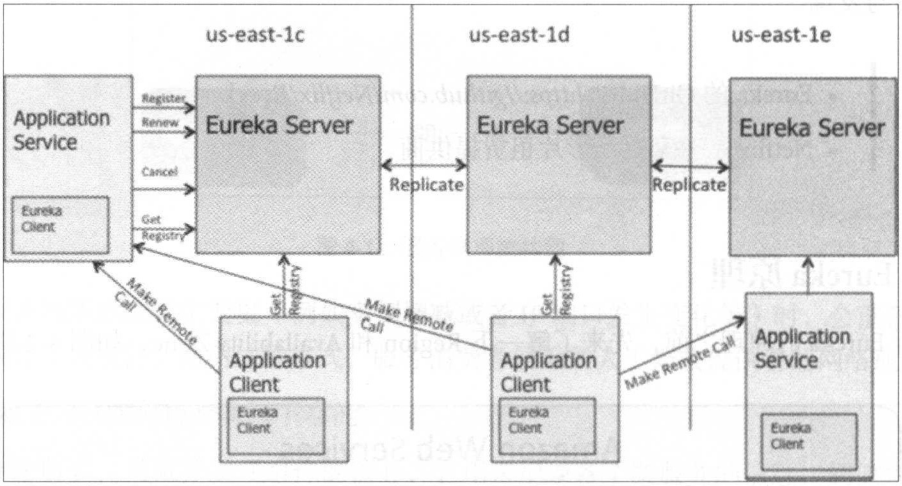


图 4-3 Eureka 架构图

图 4-3 来自 Eureka 官方的架构图，该图比较详细地描述了 Eureka 集群的工作原理。图中的组件非常多，概念也比较抽象，笔者先来用通俗易懂的文字翻译一下：

- Application Service 相当于本书中的服务提供者。
- Application Client 相当于本书中的服务消费者。
- Make Remote Call，可以理解成调用 RESTful API 的行为。
- us-east-1c、us-east-1d 等都是 zone，它们都属于 us-east-1 这个 region。

由图 4-3 可知，Eureka 包含两个组件：Eureka Server 和 Eureka Client，它们的作用如下：

- Eureka Server 提供服务发现的能力，各个微服务启动时，会向 Eureka Server 注册自己的信息（例如 IP、端口、微服务名称等），Eureka Server 会存储这些信息。
- Eureka Client 是一个 Java 客户端，用于简化与 Eureka Server 的交互。
- 微服务启动后，会周期性（默认 30 秒）地向 Eureka Server 发送心跳以续约自己的“租期”。
- 如果 Eureka Server 在一定时间内没有接收到某个微服务实例的心跳，Eureka Server 将会注销该实例（默认 90 秒）。

- 默认情况下，Eureka Server 同时也是 Eureka Client。多个 Eureka Server 实例，互相之间通过复制的方式，来实现服务注册表中数据的同步。
- Eureka Client 会缓存服务注册表中的信息。这种方式有一定的优势——首先，微服务无须每次请求都查询 Eureka Server，从而降低了 Eureka Server 的压力；其次，即使 Eureka Server 所有节点都宕掉，服务消费者依然可以使用缓存中的信息找到服务提供者并完成调用。

综上，Eureka 通过心跳检查、客户端缓存等机制，提高了系统的灵活性、可伸缩性和可用性。

4.4 编写 Eureka Server

本节来编写一个 Eureka Server。

1. 创建一个 ArtifactId 是`microservice-discovery-eureka` 的 Maven 工程，并为项目添加以下依赖。

```
<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka-server</artifactId>
  </dependency>
</dependencies>
```

2. 编写启动类，在启动类上添加`@EnableEurekaServer` 注解，声明这是一个 Eureka Server。

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaApplication {
  public static void main(String[] args) {
    SpringApplication.run(EurekaApplication.class, args);
  }
}
```

3. 在配置文件 application.yml 中添加以下内容。

```
server:
  port: 8761
eureka:
  client:
    registerWithEureka: false
```

```
    fetchRegistry: false
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

简要讲解一下 application.yml 中的配置属性：

- `eureka.client.registerWithEureka`：表示是否将自己注册到 Eureka Server，默认为 true。由于当前应用就是 Eureka Server，故而设为 false。
- `eureka.client.fetchRegistry`：表示是否从 Eureka Server 获取注册信息，默认为 true。因为这是一个单点的 Eureka Server，不需要同步其他的 Eureka Server 节点的数据，故而设为 false。
- `eureka.client.serviceUrl.defaultZone`：设置与 Eureka Server 交互的地址，查询服务和注册服务都需要依赖这个地址。默认是`http://localhost:8761/eureka`；多个地址可使用,分隔。

这样一个 Eureka Server 就编写完成了。

测试

启动 Eureka Server，访问*http://localhost:8761/*，可看到如图 4-4 所示的界面。

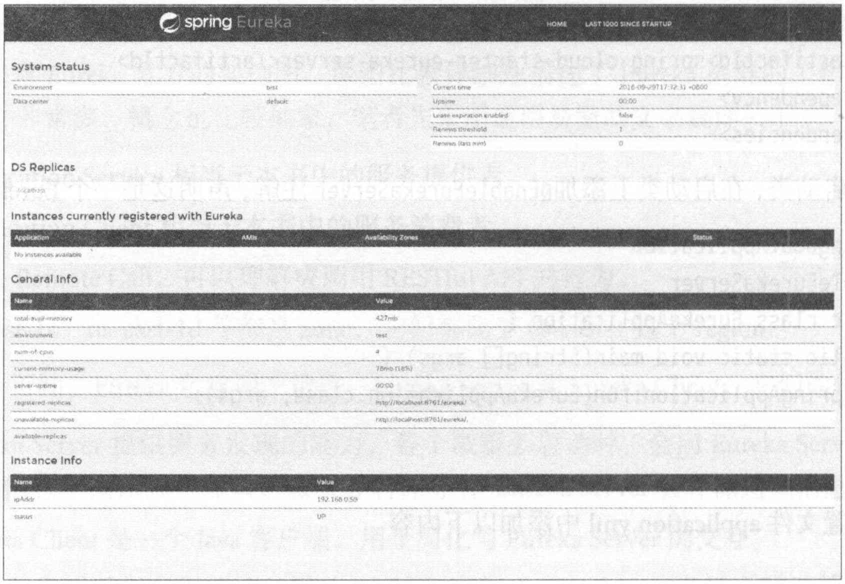


图 4-4 Eureka 首页

> 由图可知，Eureka Server 的首页展示了很多信息，例如当前实例的系统状态、注册到 Eureka Server 上的服务实例、常用信息、实例信息等。显然，当前还没有任何微服务实例被注册到 Eureka Server 上。

4.5 将微服务注册到 Eureka Server 上

本节将之前编写的用户微服务注册到 Eureka Server 上。

1. 复制项目`microservice-simple-provider-user`，将 ArtifactId 修改为 `microservice-provider-user`。

2. 在 pom.xml 中添加以下依赖。

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
```

3. 在配置文件 application.yml 中添加以下配置。

```
spring:
  application:
    name: microservice-provider-user
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
  instance:
    prefer-ip-address: true
```

   其中，`spring.application.name` 用于指定注册到 Eureka Server 上的应用名称；`eureka.instance.prefer-ip-address = true` 表示将自己的 IP 注册到 Eureka Server。如不配置该属性或将其设置为 false，则表示注册微服务所在操作系统的 hostname 到 Eureka Server。

4. 编写启动类，在启动类上添加`@EnableDiscoveryClient` 注解，声明这是一个 Eureka Client。

```
@EnableDiscoveryClient
@SpringBootApplication
public class ProviderUserApplication {
  public static void main(String[] args) {
```

```
        SpringApplication.run(ProviderUserApplication.class, args);
    }
}
```

也可以使用`@EnableEurekaClient`注解替代`@EnableDiscoveryClient`。在 Spring Cloud 中，服务发现组件有多种选择，例如 Zookeeper、Consul 等。`@EnableDiscoveryClient`为各种服务组件提供了支持，该注解是 spring-cloud-commons 项目的注解，是一个高度的抽象；而`@EnableEurekaClient`表明是 Eureka 的 Client，该注解是 spring-cloud-netflix 项目中的注解，只能与 Eureka 一起工作。当 Eureka 在项目的 classpath 中时，两个注解没有区别。

这样就可以将用户微服务注册到 Eureka Server 上了。同理，将电影微服务也注册到 Eureka Server 上，配置电影微服务的 spring.application.name 为`microservice-consumer-movie`，详见本书配套代码中的 microservice-consumer-movie 项目。

测试

1. 启动 microservice-discovery-eureka。
2. 启动 microservice-provider-user。
3. 启动 microservice-consumer-movie。
4. 访问*http://localhost:8761/*，可看到如图 4-5 的界面。

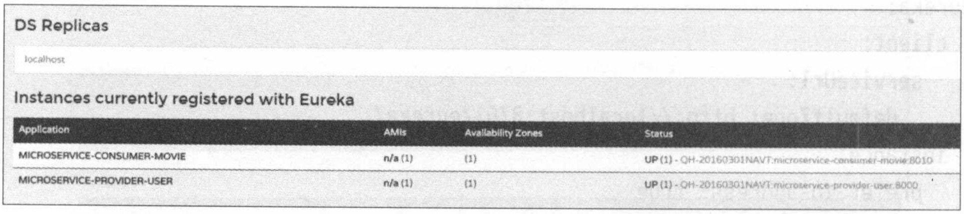


图 4-5 Eureka Server 上的微服务列表

由图可知，此时用户微服务、电影微服务已经被注册到 Eureka Server 上了。

4.6 Eureka Server 的高可用

有分布式应用开发经验的读者应该能够看出，前文编写的单节点 Eureka Server 并不适合线上生产环境。Eureka Client 会定时连接 Eureka Server，获取服务注册表中的信息并缓存在本地。微服务在消费远程 API 时总是使用本地缓存中的数据。因此一般来说，即使

Eureka Server 发生宕机，也不会影响到服务之间的调用。但如果 Eureka Server 宕机时，某些微服务也出现了不可用的情况，Eureka Client 中的缓存若不被更新，就可能会影响到微服务的调用，甚至影响到整个应用系统的高可用性。因此，在生产环境中，通常会部署一个高可用的 Eureka Server 集群。

Eureka Server 可以通过运行多个实例并相互注册的方式实现高可用部署，Eureka Server 实例会彼此增量地同步信息，从而确保所有节点数据一致。事实上，节点之间相互注册是 Eureka Server 的默认行为，还记得前文编写单节点 Eureka Server 时，额外配置了`eureka.client.registerWithEureka=false`、`eureka.client.fetchRegistry=false` 吗？

本节在前文的基础上，构建一个双节点 Eureka Server 集群。

1. 复制项目`microservice-discovery-eureka`，将 ArtifactId 修改为`microservice-discovery-eureka-ha`。
2. 配置系统的 hosts，Windows 系统的 hosts 文件路径是`C:\Windows\System32\drivers\etc\hosts`；Linux 及 Mac OS 等系统的文件路径是`/etc/hosts`。

```
127.0.0.1 peer1 peer2
```

3. 将 application.yml 修改如下：让两个节点的 Eureka Server 相互注册。

```
spring:
  application:
    name: microservice-discovery-eureka-ha
---
spring:
  # 指定profile=peer1
  profiles: peer1
server:
  port: 8761
eureka:
  instance:
    # 指定当profile=peer1时，主机名是peer1
    hostname: peer1
  client:
    serviceUrl:
      # 将自己注册到peer2这个Eureka上面去
      defaultZone: http://peer2:8762/eureka/
---
spring:
  profiles: peer2
```

```
server:
  port: 8762
eureka:
  instance:
    hostname: peer2
  client:
    serviceUrl:
      defaultZone: http://peer1:8761/eureka/
```

如上，使用连字符（---）将该 application.yml 文件分为三段。第二段和第三段分别为 spring.properties 指定了一个值，该值表示它所在的那段内容应用在哪个 Profile 里。第一段由于并未指定 spring.profiles，因此这段内容会对所有 Profile 生效。

经过以上分析，不难理解，我们定义了 peer1 和 peer2 这两个 Profile。当应用以 peer1 这个 Profile 启动时，配置该 Eureka Server 的主机名为 peer1，并将其注册到*http://peer2:8762/eureka/*；反之，当应用以 profile=peer2 时，Eureka Server 会注册到 peer1 节点的 Eureka Server。

测试

1. 打包项目，并使用以下命令启动两个 Eureka Server 节点。

   ```
   java -jar microservice-discovery-eureka-ha-0.0.1-SNAPSHOT.jar --spring.
   profiles.active=peer1
   java -jar microservice-discovery-eureka-ha-0.0.1-SNAPSHOT.jar --spring.
   profiles.active=peer2
   ```

   通过`spring.profiles.active`指定使用哪个 profile 启动。

2. 访问*http://peer1:8761*，会发现“registered-replicas”中已有 peer2 节点；同理，访问*http://peer2:8762*，也能发现其中的“registered-replicas”有 peer1 节点，如图 4-6 所示。

将应用注册到 Eureka Server 集群上

以`microservice-provider-user`项目为例，只须修改`eureka.client.serviceUrl.defaultZone`，配置多个 Eureka Server 地址，就可以将其注册到 Eureka Server 集群了。示例：

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://peer1:8761/eureka/,http://peer2:8762/eureka/
```

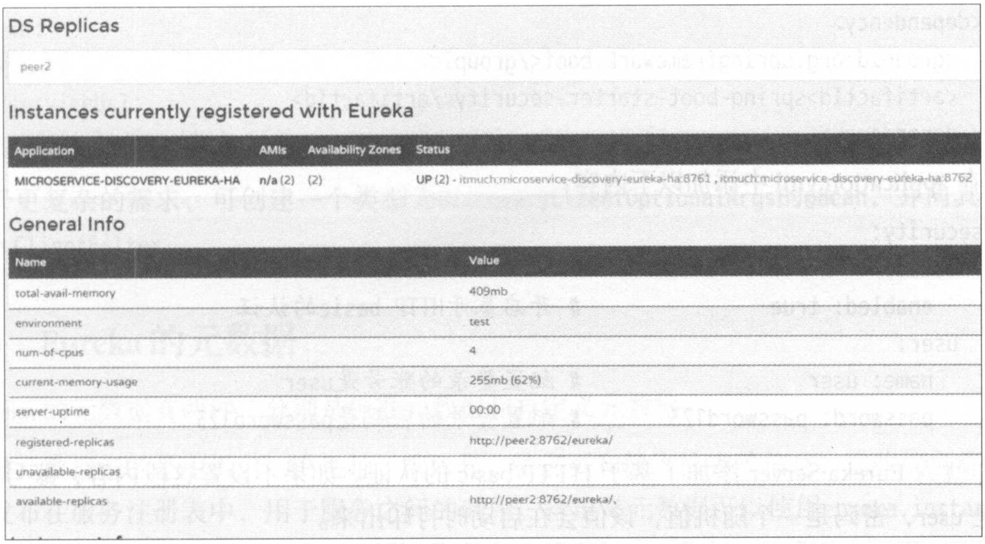


图 4-6 高可用的 Eureka 首页

这样就可以将服务注册到 Eureka Server 集群上了。

当然，微服务即使只配置 Eureka Server 集群中的某个节点，也能正常注册到 Eureka Server 集群，因为多个 Eureka Server 之间的数据会相互同步。例如：

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://peer1:8761/eureka/
```

正常情况下，这种方式与配置多个 Server 节点的效果是一样的。不过为适应某些极端场景，笔者建议在客户端配置多个 Eureka Server 节点。

4.7 为 Eureka Server 添加用户认证

在前面的示例中，Eureka Server 是允许匿名访问的，本节来构建一个需要登录才能访问的 Eureka Server。

1. 复制项目`microservice-discovery-eureka`，将 ArtifactId 修改为`microservice-discovery-eureka-authenticating`。
2. 在 pom.xml 中添加`spring-boot-starter-security`的依赖，该依赖为 Eureka Server 提供用户认证的能力。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

3. 在 application.yml 中添加以下内容：

```
security:
  basic:
    enabled: true                # 开启基于HTTP basic的认证
  user:
    name: user                   # 配置登录的账号是user
    password: password123        # 配置登录的密码是password123
```

这样就为 Eureka Server 添加了基于 HTTP basic 的认证。如果不设置这段内容，账号默认是 user，密码是一个随机值，该值会在启动时打印出来。

测试

1. 启动 microservice-discovery-eureka-authenticating。
2. 访问 *http://localhost:8761/* 需要身份验证的对话框，如图 4-7 所示。

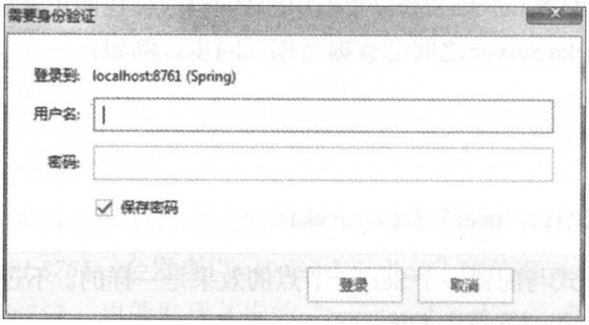


图 4-7　Eureka Server 登录界面

3. 输入账号 user、密码 password123，就可登录并访问 Eureka Server。

将微服务注册到需认证的 Eureka Server

如何才能将微服务注册到需认证的 Eureka Server 上呢？答案非常简单，只须将`eureka.client.serviceUrl.defaultZone`配置为 `http://user:password@EUREKA_HOST:EUREKA_PORT/eureka/`这种形式，就可以将 HTTP basic 认证添加到 Eureka Client 了。因此，只须稍作修改，就可以将应用注册到本例的 Eureka Server 了：

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://user:password123@localhost:8761/eureka/
```

对于更复杂的需求，可创建一个类型为DiscoveryClientOptionalArgs的@Bean，并向其中注入ClientFilter。

4.8 Eureka 的元数据

Eureka 的元数据有两种，分别是标准元数据和自定义元数据。

标准元数据指的是主机名、IP 地址、端口号、状态页和健康检查等信息，这些信息都会被发布在服务注册表中，用于服务之间的调用。自定义元数据可以使用eureka.instance.metadata-map 配置，这些元数据可以在远程客户端中访问，但一般不会改变客户端的行为，除非客户端知道该元数据的含义。

下面笔者通过一个简单的示例，帮助大家更好地理解 Eureka 的元数据。

4.8.1 改造用户微服务

1. 复制项目microservice-provider-user，将 ArtifactId 修改为microservice-provider-user-my-metadata。
2. 修改 application.yml，使用eureka.instance.metadata-map 属性为该微服务添加应自定义的元数据：

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
  instance:
    prefer-ip-address: true
    metadata-map:
      # 自定义的元数据，key/value都可以随便写。
      my-metadata: 我自定义的元数据
```

4.8.2 改造电影微服务

1. 复制项目microservice-consumer-movie，将 ArtifactId 修改为microservice-consumer-movie-understanding-metadata。

2. 修改 Controller。

```
@RestController
public class MovieController {
  @Autowired
  private RestTemplate restTemplate;

  @Autowired
  private DiscoveryClient discoveryClient;

  @GetMapping("/user/{id}")
  public User findById(@PathVariable Long id) {
    return this.restTemplate.getForObject("http://localhost:8000/" + id, User.
        class);
  }

  /**
   * 查询microservice-provider-user服务的信息并返回
   * @return microservice-provider-user服务的信息
   */
  @GetMapping("/user-instance")
  public List<ServiceInstance> showInfo() {
    return this.discoveryClient.getInstances("microservice-provider-user");
  }
}
```

使用 DiscoveryClient.getInstances(serviceId)，可查询指定微服务在 Eureka 上的实例列表。

测试

1. 启动 microservice-discovery-eureka。
2. 启动 microservice-provider-user-my-metadata。
3. 启动 microservice-consumer-movie-understanding-metadata。
4. 访问 *http://localhost:8761/eureka/apps* 可查看 Eureka 的 metadata。
5. 访问 *http://localhost:8010/user-instance*，可以返回类似如下的内容。

```
[
  {
    "host": "192.168.1.106",
```

```
        "port": 8000,
        "metadata": {
          "my-metadata": "我自定义的元数据"
        },
        "secure": false,
        "uri": "http://192.168.1.106:8000",
        "instanceInfo": {
          "instanceId": "itmuch:microservice-provider-user:8000",
          "app": "MICROSERVICE-PROVIDER-USER",
          "appGroupName": null,
          "ipAddr": "192.168.1.106",
          "sid": "na",
          "homePageUrl": "http://192.168.1.106:8000/",
          "statusPageUrl": "http://192.168.1.106:8000/info",
          "healthCheckUrl": "http://192.168.1.106:8000/health",
          "metadata": {
            "my-metadata": "我自定义的元数据"
          },
          ...
        },
        "serviceId": "MICROSERVICE-PROVIDER-USER"
      }
]
```

可以看到，使用 DiscoveryClient 的 API 获得了用户微服务的各种信息，其中包括了标准元数据和自定义元数据。例如 IP、端口等信息都是标准元数据，用于服务之间的调用；同时，自定义的元数据 my-metadata，也可通过客户端查询到，但是并不会改变客户端的行为。

4.9 Eureka Server 的 REST 端点

Eureka Server 提供了一些 REST 端点。非 JVM 的微服务可使用这些 REST 端点操作 Eureka，从而实现注册与发现。事实上，前文所讲的 Eureka Client 就是一个使用 Java 编写的、操作这些 REST 端点的类库。也可分析这些 REST 端点，编写其他语言的 Eureka Client。

表 4-1 是 Eureka 提供的 REST 端点，可以使用 XML 或者 JSON 与这些端点通信，默认是 XML。

表 4-1 Eureka 的 REST 端点一览表

Operation	HTTP action	Description
Register new application instance	POST /eureka/apps/**appID**	Input:JSON/XMLpayload HTTP Code: 204 on success
De-register application instance	DELETE /eureka/apps/**appID/instanceID**	HTTP Code: 200 on success
Send application instance heartbeat	PUT /eureka/apps/**appID/instanceID**	HTTP Code: *200 on success* 404 if **instanceID** doesn't exist
Query for all instances	GET /eureka/apps	HTTP Code: 200 on success Output:JSON/XML
Query for all **appID** instances	GET /eureka/apps/**appID**	HTTP Code: 200 on success Output:JSON/XML
Query for a specific **appID/instanceID**	GET /eureka/apps/**appID/instanceID**	HTTP Code: 200 on success Output:JSON/XML
Query for a specific **instanceID**	GET /eureka/instances/**instanceID**	HTTP Code: 200 on success Output:JSON/XML
Take instance out of service	PUT /eureka/apps/**appID/instanceID**/status?value=OUT_OF_SERVICE	HTTP Code: *200 on success* 500 on failure
Put instance back into service (remove override)	DELETE /eureka/apps/**appID/instanceID**/status?value=UP (The value=UP is optional, it is used as a suggestion for the fallback status due to removal of the override)	HTTP Code: *200 on success* 500 on failure
Update metadata	PUT /eureka/apps/**appID/instanceID**/metadata?key=value	HTTP Code: *200 on success* 500 on failure
Query for all instances under a particular **vip address**	GET /eureka/vips/**vipAddress**	*HTTP Code: 200 on success Output:JSON/XML* 404 if the **vipAddress** does not exist.
Query for all instances under a particular **secure vip address**	GET /eureka/svips/**svipAddress**	*HTTP Code: 200 on success Output:JSON/XML* 404 if the **svipAddress** does not exist.

表 4-1 中的 appID 是应用程序的名称，instanceID 是与实例相关联的唯一 ID。在 AWS 环境中，instanceID 表示微服务实例的实例 ID，在非 AWS 环境则表示实例的主机名。

示例

注册微服务到 Eureka Server 上

使用 REST 端点向 Eureka Server 注册微服务时，需要 POST 一个符合以下 XSD 的 XML（或 JSON）请求体：

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xsd:element name="instance">
    <xsd:complexType>
      <xsd:all>
        <!-- hostName in ec2 should be the public dns name, within ec2 public dns
            name will always resolve to its private IP -->
        <xsd:element name="hostName" type="xsd:string" />
        <xsd:element name="app" type="xsd:string" />
        <xsd:element name="ipAddr" type="xsd:string" />
        <xsd:element name="vipAddress" type="xsd:string" />
        <xsd:element name="secureVipAddress" type="xsd:string" />
        <xsd:element name="status" type="statusType" />
        <xsd:element name="port" type="xsd:positiveInteger" minOccurs="0" />
        <xsd:element name="securePort" type="xsd:positiveInteger" />
        <xsd:element name="homePageUrl" type="xsd:string" />
        <xsd:element name="statusPageUrl" type="xsd:string" />
        <xsd:element name="healthCheckUrl" type="xsd:string" />
        <xsd:element ref="dataCenterInfo" minOccurs="1" maxOccurs="1" />
        <!-- optional -->
        <xsd:element ref="leaseInfo" minOccurs="0" />
        <!-- optional app specific metadata -->
        <xsd:element name="metadata" type="appMetadataType" minOccurs="0" />
      </xsd:all>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="dataCenterInfo">
    <xsd:complexType>
      <xsd:all>
        <xsd:element name="name" type="dcNameType" />
        <!-- metadata is only required if name is Amazon -->
```

```
      <xsd:element name="metadata" type="amazonMetdataType" minOccurs="0" />
    </xsd:all>
  </xsd:complexType>
</xsd:element>

<xsd:element name="leaseInfo">
  <xsd:complexType>
    <xsd:all>
      <!-- (optional) if you want to change the length of lease - default if 90
          secs -->
      <xsd:element name="evictionDurationInSecs" minOccurs="0" type="
          xsd:positiveInteger" />
    </xsd:all>
  </xsd:complexType>
</xsd:element>

<xsd:simpleType name="dcNameType">
  <!-- Restricting the values to a set of value using 'enumeration' -->
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="MyOwn" />
    <xsd:enumeration value="Amazon" />
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="statusType">
  <!-- Restricting the values to a set of value using 'enumeration' -->
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="UP" />
    <xsd:enumeration value="DOWN" />
    <xsd:enumeration value="STARTING" />
    <xsd:enumeration value="OUT_OF_SERVICE" />
    <xsd:enumeration value="UNKNOWN" />
  </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="amazonMetdataType">
  <!-- From <a class="jive-link-external-small" href="http://docs.
      amazonwebservices.com/AWSEC2/latest/DeveloperGuide/index.html?AESDG-chapter
      -instancedata.html"
```

```
    target="_blank">http://docs.amazonwebservices.com/AWSEC2/latest/
        DeveloperGuide/index.html?AESDG-chapter-instancedata.html</a> -->
  <xsd:all>
    <xsd:element name="ami-launch-index" type="xsd:string" />
    <xsd:element name="local-hostname" type="xsd:string" />
    <xsd:element name="availability-zone" type="xsd:string" />
    <xsd:element name="instance-id" type="xsd:string" />
    <xsd:element name="public-ipv4" type="xsd:string" />
    <xsd:element name="public-hostname" type="xsd:string" />
    <xsd:element name="ami-manifest-path" type="xsd:string" />
    <xsd:element name="local-ipv4" type="xsd:string" />
    <xsd:element name="hostname" type="xsd:string" />
    <xsd:element name="ami-id" type="xsd:string" />
    <xsd:element name="instance-type" type="xsd:string" />
  </xsd:all>
</xsd:complexType>

<xsd:complexType name="appMetadataType">
  <xsd:sequence>
    <!-- this is optional application specific name, value metadata -->
    <xsd:any minOccurs="0" maxOccurs="unbounded" processContents="skip" />
  </xsd:sequence>
</xsd:complexType>

</xsd:schema>
```

下面来使用 REST 端点注册微服务。

1. 启动microservice-discovery-eureka 。
2. 编写一个符合上面 XSD 的 XML，命名为 rest-api-test.xml

```
<instance>
  <instanceId>itmuch:rest-api-test:9000</instanceId>
  <hostName>itmuch</hostName>
  <app>REST-API-TEST</app>
  <ipAddr>127.0.0.1</ipAddr>
  <vipAddress>rest-api-test</vipAddress>
  <secureVipAddress>rest-api-test</secureVipAddress>
  <status>UP</status>
  <port enabled="true">9000</port>
```

```
    <securePort enabled="false">443</securePort>
    <homePageUrl>http://127.0.0.1:9000/</homePageUrl>
    <statusPageUrl>http://127.0.0.1:9000/info</statusPageUrl>
    <healthCheckUrl>http://127.0.0.1:9000/health</healthCheckUrl>
    <dataCenterInfo class="com.netflix.appinfo.InstanceInfo$DefaultDataCenterInfo"
        >
      <name>MyOwn</name>
    </dataCenterInfo>
</instance>
```

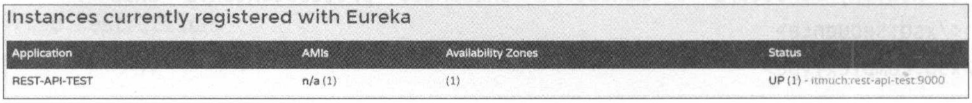
3. 使用 curl 命令测试。

```
cat ./rest-api-test.xml | curl -v -X POST -H "Content-type: application/xml" -d
    @- http://localhost:8761/eureka/apps/rest-api-test
```

终端将会输出类似以下的内容：

```
* upload completely sent off: 644 out of 644 bytes
< HTTP/1.1 204
< Content-Type: application/xml
< Date: Mon, 19 Dec 2016 05:12:21 GMT
```

此时，查看 Eureka Server 首页，会发现微服务已经成功注册，如图 4-8 所示：



Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
REST-API-TEST	n/a (1)	(1)	UP (1) - itmuch:rest-api-test:9000

图 4-8　Eureka Server 上的微服务列表

查看某微服务的所有实例

在浏览器上输入地址：*http://localhost:8761/eureka/apps/rest-api-test*，将会看到类似于如下的内容。

```
<application>
  <name>REST-API-TEST</name>
  <instance>
    <instanceId>itmuch:rest-api-test:9000</instanceId>
    <hostName>itmuch</hostName>
    <app>REST-API-TEST</app>
    <ipAddr>127.0.0.1</ipAddr>
    <status>UP</status>
    <overriddenstatus>UNKNOWN</overriddenstatus>
```

```
    <port enabled="true">9000</port>
    <securePort enabled="false">443</securePort>
    <countryId>1</countryId>
    <dataCenterInfo class="com.netflix.appinfo.InstanceInfo$DefaultDataCenterInfo">
      <name>MyOwn</name>
    </dataCenterInfo>
    <leaseInfo>
      <renewalIntervalInSecs>30</renewalIntervalInSecs>
      <durationInSecs>90</durationInSecs>
      <registrationTimestamp>1482124342257</registrationTimestamp>
      <lastRenewalTimestamp>1482124342257</lastRenewalTimestamp>
      <evictionTimestamp>0</evictionTimestamp>
      <serviceUpTimestamp>1482124341564</serviceUpTimestamp>
    </leaseInfo>
    <metadata class="java.util.Collections$EmptyMap"/>
    <homePageUrl>http://127.0.0.1:9000/</homePageUrl>
    <statusPageUrl>http://127.0.0.1:9000/info</statusPageUrl>
    <healthCheckUrl>http://127.0.0.1:9000/health</healthCheckUrl>
    <vipAddress>rest-api-test</vipAddress>
    <secureVipAddress>rest-api-test</secureVipAddress>
    <isCoordinatingDiscoveryServer>false</isCoordinatingDiscoveryServer>
    <lastUpdatedTimestamp>1482124342257</lastUpdatedTimestamp>
    <lastDirtyTimestamp>1482124341559</lastDirtyTimestamp>
    <actionType>ADDED</actionType>
  </instance>
</application>
```

从中，可以看到此微服务的所有实例，以及实例的详细信息。

注销微服务实例

将上面注册的微服务实例注销。

```
curl -v -X DELETE http://localhost:8761/eureka/apps/rest-api-test/itmuch:rest-api-
    test:9000
```

将会看到以下输出：

```
< HTTP/1.1 200
< Content-Type: application/xml
< Content-Length: 0
< Date: Tue, 20 Dec 2016 02:27:17 GMT
```

事实上，由于 rest-api-test 这个微服务并不存在，因此 Eureka Server 过一段时间后也会自动将该微服务注销。

- 本节仅挑选了两个常用端点进行测试，限于篇幅，其他端点请各位读者自行测试。
- 一些语言已有 Eureka Client 的开源实现，例如 Node.js 的 Eureka Client：*https://www.npmjs.com/package/eureka-js-client*。其他语言的 Eureka Client，读者可在 GitHub 或其他平台搜索相关实现。
- 本书使用的测试工具是 cURL，该工具在 Windows、Linux、Mac OS 均平台均有对应版本。
- 读者也可使用其他工具进行测试，例如 PostMan 等。图 4-9 是 PostMan 的测试界面，可使用它可视化地进行测试。

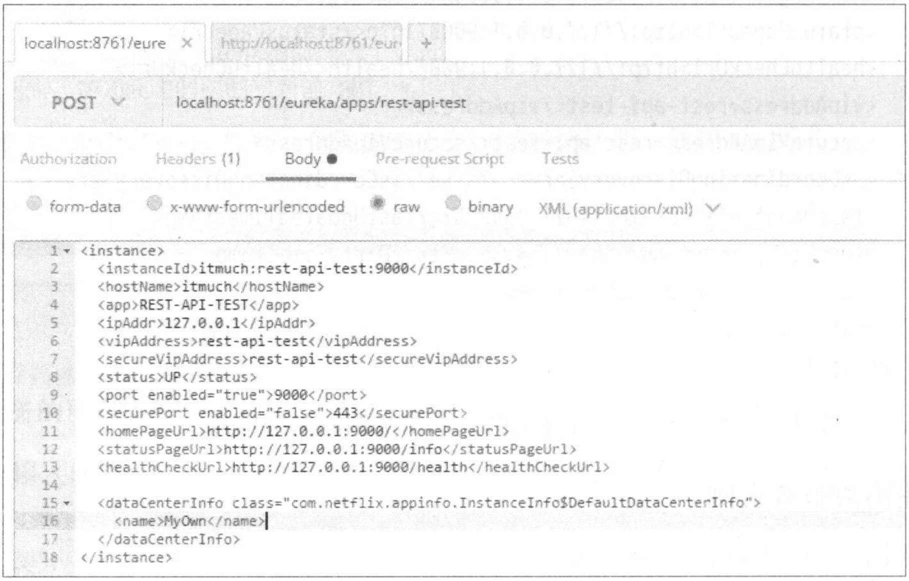


图 4-9 PostMan 测试界面

4.10　Eureka 的自我保护模式

本节来探讨 Eureka 的自我保护模式。进入自我保护模式最直观的体现，是 Eureka Server 首页输出的警告，如图 4-10 所示。

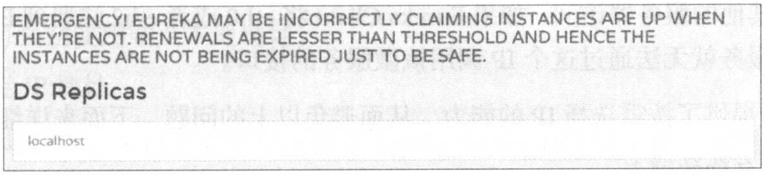


图 4-10　Eureka Server 自我保护模式界面

默认情况下，如果 Eureka Server 在一定时间内没有接收到某个微服务实例的心跳，Eureka Server 将会注销该实例（默认 90 秒）。但是当网络分区故障发生时，微服务与 Eureka Server 之间无法正常通信，以上行为可能变得非常危险了——因为微服务本身其实是健康的，此时本不应该注销这个微服务。

Eureka 通过“自我保护模式”来解决这个问题——当 Eureka Server 节点在短时间内丢失过多客户端时（可能发生了网络分区故障），那么这个节点就会进入自我保护模式。一旦进入该模式，Eureka Server 就会保护服务注册表中的信息，不再删除服务注册表中的数据（也就是不会注销任何微服务）。当网络故障恢复后，该 Eureka Server 节点会自动退出自我保护模式。

综上，自我保护模式是一种应对网络异常的安全保护措施。它的架构哲学是宁可同时保留所有微服务（健康的微服务和不健康的微服务都会保留），也不盲目注销任何健康的微服务。使用自我保护模式，可以让 Eureka 集群更加的健壮、稳定。

在 Spring Cloud 中，可以使用`eureka.server.enable-self-preservation = false`禁用自我保护模式。

- Eureka 官方关于自我保护模式的介绍：*https://github.com/Netflix/eureka/wiki/Understanding-Eureka-Peer-to-Peer-Communication*。
- Eureka 的 FAQ，其中讲到了自我保护模式：*https://github.com/Netflix/eureka/wiki/FAQ*。
- Eureka 与 Zookeeper 做服务发现的对比：*http://dockone.io/article/78*。
- Eureka 不注销任何微服务的讨论：*http://stackoverflow.com/questions/32616329/eureka-never-unregisters-a-service*。

4.11 多网卡环境下的 IP 选择

对于多网卡的服务器，各个微服务注册到 Eureka Serve 上的 IP 要如何指定呢？

指定 IP 在某些场景下很有用。例如某台服务器有 eth0、eth1 和 eth2 三块网卡，但是只有 eth1 可以被其他的服务器访问；如果 Eureka Client 将 eth0 或者 eth2 注册到 Eureka Server 上，其他微服务就无法通过这个 IP 调用该微服务的接口。

Spring Cloud 提供了按需选择 IP 的能力，从而避免以上的问题。下面来详细讨论。

1. 忽略指定名称的网卡

   例如：

```
spring:
  cloud:
    inetutils:
      ignored-interfaces:
        - docker0
        - veth.*
eureka:
  instance:
    prefer-ip-address: true
```

   这样就可以忽略 docker0 网卡以及所有以 veth 开头的网卡。

2. 使用正则表达式，指定使用的网络地址

   示例：

```
spring:
  cloud:
    inetutils:
      preferredNetworks:
        - 192.168
        - 10.0
eureka:
  instance:
    prefer-ip-address: true
```

3. 只使用站点本地地址

   示例：

```
spring:
  cloud:
```

```
    inetutils:
      useOnlySiteLocalInterfaces: true
  eureka:
    instance:
      prefer-ip-address: true
```

这样就可强制使用站点本地地址。

4. 手动指定 IP 地址

在某些极端场景下，可以手动指定注册到 Eureka Server 的微服务 IP。示例：

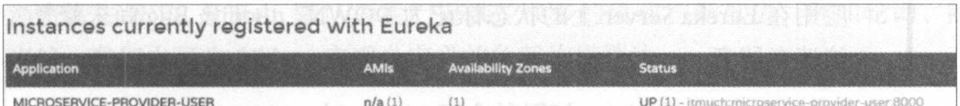
```
eureka:
  instance:
    prefer-ip-address: true
    ip-address: 127.0.0.1
```

> 本节相关代码，详见本书配套代码中的 microservice-provider-user-ip 项目。

- 站点本地地址与链路本地地址：*https://4sysops.com/archives/ipv6-tutorial-part-6-site-local-addresses-and-link-local-addresses/*。
- 源码分析：*http://www.itmuch.com/spring-cloud-code-read/spring-cloud-code-read-eureka-registry-ip/*。

4.12　Eureka 的健康检查

先来看一下 Eureka 首页，如图 4-11 所示。



Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
MICROSERVICE-PROVIDER-USER	n/a (1)	(1)	UP (1) - itmuch:microservice-provider-user:8000

图 4-11　Eureka Server 上的微服务列表

由图可见，在 Status 一栏有个 UP，表示应用程序状态正常。应用状态还有其他取值，例如 DOWN、OUT_OF_SERVICE、UNKNOWN 等。只有标记为“UP”的微服务会被请求。

前文讲过，Eureka Server 与 Eureka Client 之间使用心跳机制来确定 Eureka Client 的状态，默认情况下，服务器端与客户端的心跳保持正常，应用程序就会始终保持“UP”状态。

以上机制并不能完全反映应用程序的状态。举个例子，微服务与 Eureka Server 之间的心跳正常，Eureka Server 认为该微服务“UP”；然而，该微服务的数据源发生了问题（例如因为网络抖动，连不上数据源），根本无法正常工作。

前文说过，Spring Boot Actuator 提供了/health 端点，该端点可展示应用程序的健康信息。那么如何才能将该端点中的健康状态传播到 Eureka Server 呢？

要实现这一点，只须启用 Eureka 的健康检查。这样，应用程序就会将自己的健康状态传播到 Eureka Server。开启的方法非常简单，只须为微服务配置以下内容，就可以开启健康检查。

```
eureka:
  client:
    healthcheck:
      enabled: true
```

某些场景下，可能希望更细粒度地控制健康检查，此时可实现 `com.netflix.appinfo.HealthCheckHandler` 接口。

- `eureka.client.healthcheck.enabled=true` 只能配置在 application.yml 中，如果配置在 bootstrap.yml（后文有详解）中，可能会导致一些不良的副作用，例如应用注册到 Eureka Server 上的状态是 UNKNOWN。
- 当`eureka.client.healthcheck.enabled=true` 时，/pause 端点（该端点由 Spring Boot Actuator 提供，用于暂停应用）无法正常工作，详见：*https://github.com/spring-cloud/spring-cloud-netflix/issues/1571*，经笔者测试，发现当`eureka.client.healthcheck.enabled=true`时，请求/pause 端点无法将应用在 Eureka Server 上的状态标记为 DOWN。由于该 Bug 尚未修复，建议读者留意。

Eureka 健康检查相关博客：*https://jmnarloch.wordpress.com/2015/09/02/spring-cloud-fixing-eureka-application-status/*。

5

使用 Ribbon 实现客户端侧负载均衡

经过前文的讲解，已经实现了微服务的注册与发现。启动各个微服务时，Eureka Client 会把自己的网络信息注册到 Eureka Server 上。世界似乎更美好了一些。

然而，这样的架构依然有一些问题，比如负载均衡。一般来说，在生产环境中，各个微服务都会部署多个实例。那么服务消费者要如何将请求分摊到多个服务提供者实例上呢？

5.1 Ribbon 简介

Ribbon 是 Netflix 发布的负载均衡器，它有助于控制 HTTP 和 TCP 客户端的行为。为 Ribbon 配置服务提供者地址列表后，Ribbon 就可基于某种负载均衡算法，自动地帮助服务消费者去请求。Ribbon 默认为我们提供了很多的负载均衡算法，例如轮询、随机等。当然，我们也可为 Ribbon 实现自定义的负载均衡算法。

在 Spring Cloud 中，当 Ribbon 与 Eureka 配合使用时，Ribbon 可自动从 Eureka Server 获取服务提供者地址列表，并基于负载均衡算法，请求其中一个服务提供者实例。图 5-1 展示了 Ribbon 与 Eureka 配合使用时的大致架构。

Ribbon 的 GitHub：*https://github.com/Netflix/ribbon*。

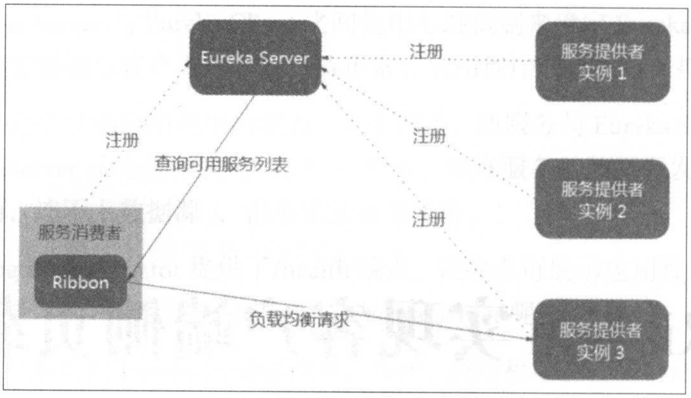


图 5-1 Eureka 与 Ribbon 配合使用架构图

5.2 为服务消费者整合 Ribbon

本节来为前文编写的电影微服务整合 Ribbon。

1. 复制项目`microservice-consumer-movie`，将 ArtifactId 修改为`microservice-consumer-movie-ribbon`。
2. 为项目引入 Ribbon 的依赖，Ribbon 的依赖是：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-ribbon</artifactId>
</dependency>
```

由于前文中已为电影微服务添加了`spring-cloud-starter-eureka`，该依赖已经包含了`spring-cloud-starter-ribbon`，所以无须再次引入。

3. 为 RestTemplate 添加`@LoadBalanced`注解。

```
@Bean
@LoadBalanced
public RestTemplate restTemplate() {
  return new RestTemplate();
}
```

回顾以前的代码，使用如下方法实例化 RestTemplate：

```
@Bean
public RestTemplate restTemplate() {
  return new RestTemplate();
}
```

两者对比可以发现，只须添加注解@LoadBalanced 注解，就可为 RestTemplate 整合 Ribbon，使其具备负载均衡的能力。

4. 将 Controller 代码修改如下：

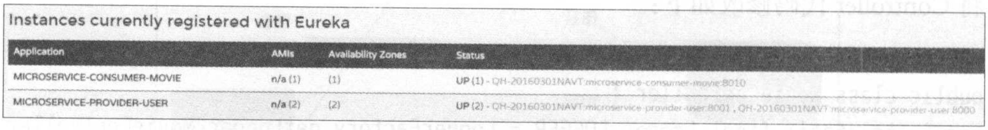
```
@RestController
public class MovieController {
  private static final Logger LOGGER = LoggerFactory.getLogger(MovieController.
      class);
  @Autowired
  private RestTemplate restTemplate;
  @Autowired
  private LoadBalancerClient loadBalancerClient;

  @GetMapping("/user/{id}")
  public User findById(@PathVariable Long id) {
    return this.restTemplate.getForObject("http://microservice-provider-user/" +
        id, User.class);
  }

  @GetMapping("/log-instance")
  public void logUserInstance() {
    ServiceInstance serviceInstance = this.loadBalancerClient.choose("
        microservice-provider-user");
    // 打印当前选择的是哪个节点
    MovieController.LOGGER.info("{}:{}:{}", serviceInstance.getServiceId(),
        serviceInstance.getHost(), serviceInstance.getPort());
  }
}
```

由代码可知，我们将请求的地址改成了`http://microservice-provider-user/`。`microservice-provider-user`是用户微服务的**虚拟主机名**（virtual host name），当 Ribbon 和 Eureka 配合使用时，会自动将虚拟主机名映射成微服务的网络地址。在新增的`logUserInstance()`方法中可使用 LoadBalancerClient 的 API 更加直观地获取当前选择的用户微服务节点。

测试

1. 启动 microservice-discovery-eureka。
2. 启动 2 个或更多 microservice-provider-user 实例。

3. 启动 microservice-consumer-movie-ribbon。
4. 访问*http://localhost:8761*，结果如图 5-2 所示。



Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
MICROSERVICE-CONSUMER-MOVIE	n/a (1)	(1)	UP (1) - QH-20160301NAVT:microservice-consumer-movie:8010
MICROSERVICE-PROVIDER-USER	n/a (2)	(2)	UP (2) - QH-20160301NAVT:microservice-provider-user:8001 , QH-20160301NAVT:microservice-provider-user:8000

图 5-2 Eureka Server 上的微服务列表

5. 多次访问 http://localhost:8010/user/1，返回如下结果：

```
{
    "id": 1,
    "username": "account1",
    "name": "张三",
    "age": 20,
    "balance": 100
}
```

同时，两个用户微服务实例都会打印查询如下日志：

```
Hibernate: select user0_.id as id1_0_0_, user0_.age as age2_0_0_, user0_.
    balance as balance3_0_0_, user0_.name as name4_0_0_, user0_.username as
    username5_0_0_ from user user0_ where user0_.id=?
...
```

6. 多次访问*http://localhost:8010/log-user-instance*，控制台会打印如下的日志：

```
2016-11-04 17:17:10.839  INFO 28596 --- [nio-8010-exec-9] c.i.c.s.user.con
troller.MovieController  : microservice-provider-user:192.168.0.59:8001
2016-11-04 17:17:11.068  INFO 28596 --- [io-8010-exec-10] c.i.c.s.user.con
troller.MovieController  : microservice-provider-user:192.168.0.59:8000
2016-11-04 17:17:11.258  INFO 28596 --- [nio-8010-exec-1] c.i.c.s.user.con
troller.MovieController  : microservice-provider-user:192.168.0.59:8001
2016-11-04 17:17:11.496  INFO 28596 --- [nio-8010-exec-2] c.i.c.s.user.con
troller.MovieController  : microservice-provider-user:192.168.0.59:8000
2016-11-04 17:17:11.688  INFO 28596 --- [nio-8010-exec-3] c.i.c.s.user.con
troller.MovieController  : microservice-provider-user:192.168.0.59:8001
2016-11-04 17:17:12.015  INFO 28596 --- [nio-8010-exec-4] c.i.c.s.user.con
troller.MovieController  : microservice-provider-user:192.168.0.59:8000
```

可以看到，此时请求会均匀分布到两个用户微服务节点上，说明已经实现了负载均衡。

- 虚拟主机名与虚拟 IP 非常类似，如果大家接触过 HAProxy 或 Heartbeat，理解虚拟主机名就非常容易了。如果无法理解虚拟主机名，可将其简单理解成为提供者的服务名称，因为在默认情况下，虚拟主机名和服务名称是一致的。当然，也可使用配置属性`eureka.instance.virtual-host-name`或者`eureka.instance.secure-virtual-host-name`指定虚拟主机名。
- 不能将`restTemplate.getForObject(...)` 与`loadBalancerClient.choose(...)`写在同一个方法中，两者之间会有冲突——因为此时代码中的 `restTemplate` 实际上是一个 Ribbon 客户端，本身已经包含了“choose”的行为。

虚拟主机名不能包含“_”之类的字符，否则 Ribbon 在调用时会报异常。相关 Issue 详见：*https://github.com/spring-cloud/spring-cloud-netflix/issues/1582*。

5.3 使用 Java 代码自定义 Ribbon 配置

很多场景下，可能根据需要自定义 Ribbon 的配置，例如修改 Ribbon 的负载均衡规则等。Spring Cloud Camden 允许使用 Java 代码或属性自定义 Ribbon 的配置，两种方式是等价的。

本节来讨论如何使用 Java 代码自定义 Ribbon 的配置。

在 Spring Cloud 中，Ribbon 的默认配置如下 (格式是`BeanType` beanName: `ClassName`)：

- `IClientConfig` ribbonClientConfig: `DefaultClientConfigImpl`
- `IRule` ribbonRule: `ZoneAvoidanceRule`
- `IPing` ribbonPing: `NoOpPing`
- `ServerList` ribbonServerList: `ConfigurationBasedServerList`
- `ServerListFilter` ribbonServerListFilter: `ZonePreferenceServerListFilter`
- `ILoadBalancer` ribbonLoadBalancer: `ZoneAwareLoadBalancer`

简单说明一下，例如以下代码：

```
@Bean
@ConditionalOnMissingBean
public IRule ribbonRule(IClientConfig config) {
  ZoneAvoidanceRule rule = new ZoneAvoidanceRule();
  rule.initWithNiwsConfig(config);
  return rule;
}
// 来自org.springframework.cloud.netflix.ribbon.RibbonClientConfiguration
```

BeanType 是 IRule，beanName 是 ribbonRule，ClassName 是 ZoneAvoidanceRule，这是一种根据服务提供者所在 Zone 的性能以及服务提供者可用性综合计算，选择提供者节点的负载均衡规则。

在 Spring Cloud 中，Ribbon 默认的配置类是RibbonClientConfiguration。也可使用一个 POJO 自定义 Ribbon 的配置（自定义配置会覆盖默认配置）。这种配置是细粒度的，不同的 Ribbon 客户端可以使用不同的配置。

下面来为名为 microservice-provider-user 的 Ribbon 客户端自定义配置。

1. 复制项目microservice-consumer-movie-ribbon，将 ArtifactId 修改为 microservice-consumer-movie-ribbon-customizing。
2. 创建 Ribbon 的配置类。

```
/**
 * 该类为Ribbon的配置类
 * 注意：该类不应该在主应用程序上下文的@ComponentScan 中。
 * @author 周立
 */
@Configuration
public class RibbonConfiguration {
  @Bean
  public IRule ribbonRule() {
    // 负载均衡规则，改为随机
    return new RandomRule();
  }
}
```

3. 创建一个空类，并在其上添加 @Configuration 注解和 @RibbonClient 注解。

```
/**
 * 使用RibbonClient，为特定name的Ribbon Client自定义配置。
 * 使用@RibbonClient的configuration属性，指定Ribbon的配置类。
 * 可参考的示例：
 * http://spring.io/guides/gs/client-side-load-balancing/
 * @author 周立
 */
@Configuration
@RibbonClient(name = "microservice-provider-user", configuration =
  RibbonConfiguration.class)
public class TestConfiguration {
}
```

由代码可知，使用 @RibbonClient 注解的 configuration 属性，即可自定义指定名称 Ribbon 客户端的配置。

测试

1. 启动 microservice-discovery-eureka。
2. 启动 2 个或更多 microservice-provider-user 实例。
3. 启动 microservice-consumer-movie-ribbon-customizing。
4. 多次访问*http://localhost:8010/log-user-instance*，可获得类似于如下的日志：

```
2016-11-01 10:56:44.717  INFO 5624 --- [nio-8010-exec-2] c.i.c.study.user.
service.MovieService      : microservice-provider-user:192.168.0.59:8000
2016-11-01 10:56:46.657  INFO 5624 --- [nio-8010-exec-6] c.i.c.study.user.
service.MovieService      : microservice-provider-user:192.168.0.59:8000
2016-11-01 10:56:46.849  INFO 5624 --- [io-8010-exec-10] c.i.c.study.user.
service.MovieService      : microservice-provider-user:192.168.0.59:8001
2016-11-01 10:56:46.996  INFO 5624 --- [nio-8010-exec-9] c.i.c.study.user.
service.MovieService      : microservice-provider-user:192.168.0.59:8000
2016-11-01 10:56:47.148  INFO 5624 --- [nio-8010-exec-7] c.i.c.study.user.
service.MovieService      : microservice-provider-user:192.168.0.59:8001
2016-11-01 10:56:47.312  INFO 5624 --- [nio-8010-exec-2] c.i.c.study.user.
service.MovieService      : microservice-provider-user:192.168.0.59:8001
```

此时请求会随机分布到两个用户微服务节点上，说明已经实现了 Ribbon 的自定义配置。

> 必须注意的是，本例中的RibbonConfiguration 类不能包含在主应用程序上下文的 @ComponentScan 中，否则该类中的配置信息就被所有的@RibbonClient 共享。
>
> 因此，如果只想自定义某一个 Ribbon 客户端的配置，必须防止@Configuration 注解的类所在的包与 @ComponentScan 扫描的包重叠，或应显式指定 @ComponentScan 不扫描 @Configuration 类所在的包。

5.4 使用属性自定义 Ribbon 配置

从 Spring Cloud Netflix 1.2.0 开始（Spring Cloud Camden SR4 使用的版本是 1.2.4），Ribbon 支持使用属性自定义 Ribbon 客户端。这种方式比使用 Java 代码配置的方式更加方便。

支持的属性如下，配置的前缀是 <clientName>.ribbon.。

- NFLoadBalancerClassName：配置 ILoadBalancer 的实现类
- NFLoadBalancerRuleClassName：配置IRule 的实现类
- NFLoadBalancerPingClassName：配置IPing 的实现类
- NIWSServerListClassName：配置 ServerList 的实现类
- NIWSServerListFilterClassName：配置ServerListFilter 的实现类

下面采用属性来修改名为microservice-provider-user 的 Ribbon 客户端的负载均衡规则。

复制项目microservice-consumer-movie-ribbon，将 ArtifactId 修改为microservice-consumer-movie-ribbon-customizing-properties。在项目的 application.yml 中添加以下内容即可：

```
microservice-provider-user:
  ribbon:
    NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule
```

这样，就可以将负载均衡规则修改为随机。由代码不难看出，使用属性自定义的方式比用 Java 代码配置方便很多。

测试

1. 启动 microservice-discovery-eureka。
2. 启动两个或更多 microservice-provider-user 实例。
3. 启动 microservice-consumer-movie-ribbon-customizing-properties。

4. 多次访问*http://localhost:8010/log-user-instance*，查看日志，会发现此时请求依然会随机分布到两个用户微服务节点上。

5.5　脱离 Eureka 使用 Ribbon

在前文的示例中，是将 Ribbon 与 Eureka 配合使用的。但现实中可能不具备这样的条件，例如一些遗留的微服务，它们可能并没有注册到 Eureka Server 上，甚至根本不是使用 Spring Cloud 开发的，此时要想使用 Ribbon 实现负载均衡，要怎么办呢？

Ribbon 支持脱离 Eureka 使用，此时，架构如图 5-3 所示。

图 5-3　脱离 Eureka 使用 Ribbon 架构图

下面笔者通过一个简单的示例为大家讲解如何脱离 Eureka 使用 Ribbon。

1. 复制项目`microservice-consumer-movie-ribbon`，将 ArtifactId 修改为 `microservice-consumer-movie-without-eureka`。
2. 为了让测试更具说服力，干脆为项目去掉 Eureka 的依赖`spring-cloud-starter-eureka`，只使用 Ribbon 的依赖`spring-cloud-starter-ribbon`。在项目的 pom.xml 中，找到：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
```

修改为：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-ribbon</artifactId>
</dependency>
```

3. 去掉启动类上的`@EnableDiscoveryClient`注解。
4. 将 application.yml 改成如下：

```
server:
  port: 8010
spring:
  application:
    name: microservice-consumer-movie
microservice-provider-user:
  ribbon:
    listOfServers: localhost:8000,localhost:8001
```

其中，属性`microservice-provider-user.ribbon.listOfServers`用于为名为`microservice-provider-user`的 Ribbon 客户端设置请求的地址列表。

测试

1. 启动两个或更多 microservice-simple-provider-user 实例。

```
java -jar microservice-simple-provider-user-0.0.1-SNAPSHOT.jar
java -jar microservice-simple-provider-user-0.0.1-SNAPSHOT.jar --server.port
=8001
```

2. 启动 microservice-consumer-movie-without-eureka。
3. 多次访问*http://localhost:8010/log-user-instance*，控制台打印如下日志。

```
2016-11-09 17:56:32.223  INFO 21844 --- [nio-8010-exec-2] c.i.c.s.user.con
troller.MovieController  : microservice-provider-user:localhost:8001
2016-11-09 17:56:32.657  INFO 21844 --- [nio-8010-exec-3] c.i.c.s.user.con
troller.MovieController  : microservice-provider-user:localhost:8000
2016-11-09 17:56:33.121  INFO 21844 --- [nio-8010-exec-4] c.i.c.s.user.con
troller.MovieController  : microservice-provider-user:localhost:8001
2016-11-09 17:56:33.555  INFO 21844 --- [nio-8010-exec-5] c.i.c.s.user.con
troller.MovieController  : microservice-provider-user:localhost:8000
2016-11-09 17:56:34.015  INFO 21844 --- [nio-8010-exec-6] c.i.c.s.user.con
troller.MovieController  : microservice-provider-user:localhost:8001
2016-11-09 17:56:38.842  INFO 21844 --- [nio-8010-exec-7] c.i.c.s.user.con
troller.MovieController  : microservice-provider-user:localhost:8000
```

由结果可知，尽管电影微服务和用户微服务此时并没有注册到 Eureka 上，Ribbon 仍可正常工作，请求依旧会分摊到两个用户微服务节点上。

第6章 使用 Feign 实现声明式 REST 调用

前文的示例中是使用 RestTemplate 实现 REST API 调用的，代码大致如下：

```
public User findById(Long id) {
  return this.ribbonRestTemplate.getForObject("http://microservice-provider-user/"
      + id, User.class);
}
```

由代码可知，我们是使用拼接字符串的方式构造 URL 的，该 URL 只有一个参数。然而在现实中，URL 中往往有多个参数。如果这时还使用这种方式构造 URL，那么就会变得很低效，并且难以维护。举个例子，想要请求这样的 URL：

```
http://localhost:8010/search?name=张三&username=account1&age=20
```

如使用拼接字符串的方式构建请求 URL，那么代码可编写如下：

```
public User[] findById(String name, String username, Integer age) {
  Map<String, Object> paramMap = Maps.newHashMap();
  paramMap.put("name", name);
  paramMap.put("username", username);
  paramMap.put("age", age);
  return this.restTemplate.getForObject("http://microservice-provider-user/search?
      name={name}&username={username}&age={age}", User[].class,
```

```
    paramMap);
}
```

在这里，URL 仅包含 3 个参数。如果 URL 更加复杂，例如有 10 个以上的参数，那么代码会变得难以维护。

如何解决这种问题呢？读者可带着这个疑问阅读本章。

6.1 Feign 简介

Feign 是 Netflix 开发的声明式、模板化的 HTTP 客户端，其灵感来自 Retrofit、JAXRS-2.0 以及 WebSocket。Feign 可帮助我们更加便捷、优雅地调用 HTTP API。

在 Spring Cloud 中，使用 Feign 非常简单——创建一个接口，并在接口上添加一些注解，代码就完成了。Feign 支持多种注解，例如 Feign 自带的注解或者 JAX-RS 注解等。

Spring Cloud 对 Feign 进行了增强，使 Feign 支持了 Spring MVC 注解，并整合了 Ribbon 和 Eureka，从而让 Feign 的使用更加方便。

> Feign 的 GitHub：*https://github.com/OpenFeign/feign*。

6.2 为服务消费者整合 Feign

前文电影微服务是使用 RestTemplate（负载均衡通过整合 Ribbon 实现）调用 RESTful API 的。本节让电影微服务使用 Feign，实现声明式的 RESTful API 调用。

1. 复制项目`microservice-consumer-movie`，将 ArtifactId 修改为`microservice-consumer-movie-feign`。
2. 添加 Feign 的依赖。

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-feign</artifactId>
</dependency>
```

3. 创建一个 Feign 接口，并添加 @FeignClient 注解。

```
@FeignClient(name = "microservice-provider-user")
public interface UserFeignClient {
```

```
  @RequestMapping(value = "/{id}", method = RequestMethod.GET)
  public User findById(@PathVariable("id") Long id);
}
```

@FeignClient注解中的microservice-provider-user是一个任意的客户端名称，用于创建 Ribbon 负载均衡器。在本例中，由于使用了 Eureka，所以 Ribbon 会把microservice-provider-user解析成 Eureka Server 服务注册表中的服务。当然，如果不想使用 Eureka，可使用service.ribbon.listOfServers属性配置服务器列表（详见 5.5 节）。

还可使用 url 属性指定请求的 URL（URL 可以是完整的 URL 或者主机名），例如 @FeignClient(name = "microservice-provider-user", url = "http://localhost:8000/")。

4. 修改 Controller 代码，让其调用 Feign 接口。

```
@RestController
public class MovieController {
  @Autowired
  private UserFeignClient userFeignClient;

  @GetMapping("/user/{id}")
  public User findById(@PathVariable Long id) {
    return this.userFeignClient.findById(id);
  }
}
```

5. 修改启动类，为其添加@EnableFeignClients注解，如下：

```
@EnableDiscoveryClient
@SpringBootApplication
@EnableFeignClients
public class ConsumerMovieApplication {
  public static void main(String[] args) {
    SpringApplication.run(ConsumerMovieApplication.class, args);
  }
}
```

这样，电影微服务就可以用 Feign 去调用用户微服务的 API 了。

测试

1. 启动 microservice-discovery-eureka。
2. 启动 2 个或更多 microservice-provider-user 实例。

3. 启动 microservice-consumer-movie-feign。
4. 多次访问 http://localhost:8010/user/1，返回如下结果。

```
{
  "id": 1,
  "username": "account1",
  "name": "张三",
  "age": 20,
  "balance": 100
}
```

两个用户微服务实例都会打印类似如下的日志。

```
Hibernate: select user0_.id as id1_0_0_, user0_.age as age2_0_0_, user0_.
  balance as balance3_0_0_, user0_.name as name4_0_0_, user0_.username as
  username5_0_0_ from user user0_ where user0_.id=?
...
```

以上结果说明，我们不但实现了声明式的 REST API 调用，同时还实现了客户端侧的负载均衡。

6.3 自定义 Feign 配置

本节来讨论如何自定义 Feign 的配置。

在 Spring Cloud 中，Feign 的默认配置类是FeignClientsConfiguration，该类定义了 Feign 默认使用的编码器、解码器、所使用的契约等。

Spring Cloud 允许通过注解@FeignClient的 configuration 属性自定义 Feign 的配置，自定义配置的优先级比 FeignClientsConfiguration 要高。

在 Spring Cloud 文档中可看到以下段落，描述了 Spring Cloud 提供的默认配置。另外，有的配置尽管没有提供默认值，但是 Spring 也会扫描其中列出的类型（也就是说，这部分配置也能自定义）。

> Spring Cloud Netflix provides the following beans by default for feign (BeanType beanName: ClassName):
>
> - Decoder feignDecoder: ResponseEntityDecoder (which wraps a SpringDecoder)
> - Encoder feignEncoder: SpringEncoder

- `Logger` feignLogger: `Slf4jLogger`
- `Contract` feignContract: `SpringMvcContract`
- `Feign.Builder` feignBuilder: `HystrixFeign.Builder`
- `Client` feignClient: if Ribbon is enabled it is a `LoadBalancerFeignClient`, otherwise the default feign client is used.

The OkHttpClient and ApacheHttpClient feign clients can be used by setting `feign.okhttp.enabled` or `feign.httpclient.enabled`to `true`, respectively, and having them on the classpath.

Spring Cloud Netflix *does not* provide the following beans by default for feign, but still looks up beans of these types from the application context to create the feign client:

- `Logger.Level`
- `Retryer`
- `ErrorDecoder`
- `Request.Options`
- `Collection<RequestInterceptor>`

由此可知，在 Spring Cloud 中，Feign 默认使用的契约是`SpringMvcContract`，因此它可以使用 Spring MVC 的注解。下面来自定义 Feign 的配置，让它使用 Feign 自带的注解进行工作。

1. 复制项目`microservice-consumer-movie-feign`，将 ArtifactId 修改为 `microservice-consumer-movie-feign-customizing`。
2. 创建 Feign 的配置类。

```
/**
 * 该类为Feign的配置类
 * 注意：该不应该在主应用程序上下文的@ComponentScan中。
 * @author 周立
 */
@Configuration
public class FeignConfiguration {
  /**
   * 将契约改为feign原生的默认契约。这样就可以使用feign自带的注解了。
   * @return 默认的feign契约
```

```
  */
  @Bean
  public Contract feignContract() {
    return new feign.Contract.Default();
  }
}
```

3. Feign 接口修改为如下，使用`@FeignClient`的 configuration 属性指定配置类，同时，将 findById 上的 Spring MVC 注解修改为 Feign 自带的注解。

```
/**
 * 使用@FeignClient的configuration属性，指定feign的配置类。
 * @author 周立
 */
@FeignClient(name = "microservice-provider-user", configuration =
    FeignConfiguration.class)
public interface UserFeignClient {
  /**
   * 使用feign自带的注解@RequestLine
   * @see https://github.com/OpenFeign/feign
   * @param id 用户id
   * @return 用户信息
   */
  @RequestLine("GET /{id}")
  public User findById(@Param("id") Long id);
}
```

类似地，还可自定义 Feign 的编码器、解码器、日志打印，甚至为 Feign 添加拦截器。例如，一些接口需要进行基于 Http Basic 的认证后才能调用，配置类可以这样写：

```
@Configuration
public class FooConfiguration {
  @Bean
  public BasicAuthRequestInterceptor basicAuthRequestInterceptor() {
    return new BasicAuthRequestInterceptor("user", "password");
  }
}
```

测试

1. 启动 microservice-discovery-eureka。

2. 启动 microservice-provider-user。
3. 启动 microservice-consumer-movie-feign-customizing。
4. 访问 *http://localhost:8010/user/1*，可获得如下结果。

```
{
  "id": 1,
  "username": "account1",
  "name": "张三",
  "age": 20,
  "balance": 100
}
```

说明已实现 Feign 配置的自定义。

> 和 Ribbon 配置自定义一样，本例中的`FeignConfiguration`类也不能包含在主应用程序上下文的`@ComponentScan`中，否则该类中的配置信息就会被所有的`@FeignClient`共享。
>
> 因此，如果只想自定义某个 Feign 客户端的配置，必须防止`@Configuration`注解的类所在的包与`@ComponentScan`扫描的包重叠，或应显式指定`@ComponentScan`不扫描`@Configuration`类所在的包。

6.4 手动创建 Feign

在某些场景下，前文自定义 Feign 的方式满足不了需求，此时可使用 Feign Builder API（*https://github.com/OpenFeign/feign/#basics*）手动创建 Feign。

本节围绕以下场景，为大家讲解如何手动创建 Feign。

- 用户微服务的接口需要登录后才能调用，并且对于相同的 API，不同角色的用户有不同的行为。
- 让电影微服务中的同一个 Feign 接口，使用不同的账号登录，并调用用户微服务的接口。

6.4.1 修改用户微服务

首先来改写用户微服务。

1. 复制项目microservice-provider-user，将 ArtifactId 修改为microservice-provider-user-with-auth。
2. 为项目添加以下依赖。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

3. 创建 Spring Security 的配置类。

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

  @Override
  protected void configure(HttpSecurity http) throws Exception {
    // 所有的请求，都需要经过HTTP basic认证
    http.authorizeRequests().anyRequest().authenticated().and().httpBasic();
  }

  @Bean
  public PasswordEncoder passwordEncoder() {
    // 明文编码器。这是一个不做任何操作的密码编码器，是Spring提供给我们做
        明文测试的
    // A password encoder that does nothing. Useful for testing where working
        with plain text
    return NoOpPasswordEncoder.getInstance();
  }

  @Autowired
  private CustomUserDetailsService userDetailsService;

  @Override
  protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.userDetailsService(this.userDetailsService).passwordEncoder(this.
```

```
        passwordEncoder());
  }

  @Component
  class CustomUserDetailsService implements UserDetailsService {
    /**
     * 模拟两个账户：
     * ① 账号是user，密码是password1，角色是user-role
     * ② 账号是admin，密码是password2，角色是admin-role
     */
    @Override
    public UserDetails loadUserByUsername(String username) throws
        UsernameNotFoundException {
      if ("user".equals(username)) {
        return new SecurityUser("user", "password1", "user-role");
      } else if ("admin".equals(username)) {
        return new SecurityUser("admin", "password2", "admin-role");
      } else {
        return null;
      }
    }
  }

  class SecurityUser implements UserDetails {
    private static final long serialVersionUID = 1L;

    public SecurityUser(String username, String password, String role) {
      super();
      this.username = username;
      this.password = password;
      this.role = role;
    }

    public SecurityUser() {
    }

    private Long id;
    private String username;
    private String password;
```

```
    private String role;

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
      Collection<GrantedAuthority> authorities = new ArrayList<GrantedAuthority
          >();
      SimpleGrantedAuthority authority = new SimpleGrantedAuthority(this.role);
      authorities.add(authority);
      return authorities;
    }
    ...
    // 省略空实现UserDetails类的方法、getters、setters
  }
}
```

代码中模拟了两个账号：user 和 admin，它们的密码分别是 password1 和 password2，角色分别是 user-role 和 admin-role。

4. 修改 Controller，在其中打印当前登录的用户信息。

```
@RestController
public class UserController {
  @Autowired
  private UserRepository userRepository;
  private static final Logger LOGGER = LoggerFactory.getLogger(UserController.
      class);

  @GetMapping("/{id}")
  public User findById(@PathVariable Long id) {
    Object principal = SecurityContextHolder.getContext().getAuthentication().
        getPrincipal();
    if (principal instanceof UserDetails) {
      UserDetails user = (UserDetails) principal;
      Collection<? extends GrantedAuthority> collection = user.getAuthorities();
      for (GrantedAuthority c : collection) {
        // 打印当前登录用户的信息
        UserController.LOGGER.info("当前用户是{}，角色是{}", user.getUsername
            (), c.getAuthority());
      }
    } else {
      // do other things
```

```
    }
    User findOne = this.userRepository.findOne(id);
    return findOne;
  }
}
```

用户微服务测试

1. 启动 microservice-discovery-eureka。
2. 启动 microservice-provider-user-with-auth。
3. 访问*http://localhost:8000/1*，将会弹出登录对话框，如图 6-1 所示。

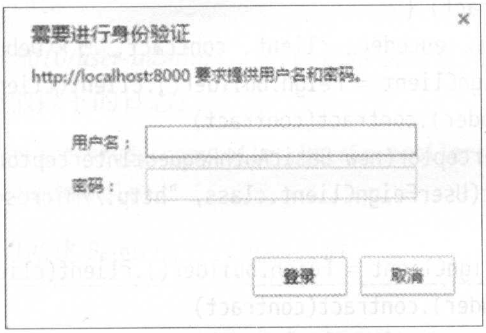


图 6-1　用户微服务登录对话框

4. 使用 user/password1 登录，可看到类似如下的日志。

```
[nio-8000-exec-5] c.i.c.study.controller.UserController     : 当前用户是user，角
    色是user-role
```

5. 使用 admin/password2 登录，可看到类似如下的日志。

```
[nio-8000-exec-9] c.i.c.study.controller.UserController     : 当前用户是admin，
    角色是admin-role
```

6.4.2 修改电影微服务

修改好用户微服务后，接下来修改电影微服务。

1. 复制项目`microservice-consumer-movie-feign`，将 ArtifactId 修改为`microservice-consumer-movie-feign-manual`。
2. 去掉 Feign 接口`UserFeignClient`上的`@FeignClient`注解

3. 去掉启动类上的@EnableFeignClients 注解。
4. 修改 Controller 如下：

```
@Import(FeignClientsConfiguration.class)
@RestController
public class MovieController {
  private UserFeignClient userUserFeignClient;

  private UserFeignClient adminUserFeignClient;

  @Autowired
  public MovieController(Decoder decoder, Encoder encoder, Client client,
      Contract contract) {
    // 这边的decoder、encoder、client、contract，可以Debug看看是什么实例
    this.userUserFeignClient = Feign.builder().client(client).encoder(encoder).
        decoder(decoder).contract(contract)
        .requestInterceptor(new BasicAuthRequestInterceptor("user", "password1")
            ).target(UserFeignClient.class, "http://microservice-provider-user/"
            );
    this.adminUserFeignClient = Feign.builder().client(client).encoder(encoder).
        decoder(decoder).contract(contract)
        .requestInterceptor(new BasicAuthRequestInterceptor("admin", "password2"
            ))
        .target(UserFeignClient.class, "http://microservice-provider-user/");
  }

  @GetMapping("/user-user/{id}")
  public User findByIdUser(@PathVariable Long id) {
    return this.userUserFeignClient.findById(id);
  }

  @GetMapping("/user-admin/{id}")
  public User findByIdAdmin(@PathVariable Long id) {
    return this.adminUserFeignClient.findById(id);
  }
}
```

其中，@Import 导入的FeignClientsConfiguration 是 Spring Cloud 为 Feign 默认提供的配置类。

userUserFeignClient 登录账号 user，adminUserFeignClient 登录账号 admin，它们使用的是同一个接口 UserFeignClient。

测试

1. 启动 microservice-discovery-eureka。
2. 启动 microservice-provider-user-with-auth。
3. 启动 microservice-consumer-movie-feign-manual。
4. 访问*http://localhost:8010/user-user/1*，可以正常获得查询结果，同时可以看到用户微服务打印类似以下的日志：

   ```
   [nio-8000-exec-5] c.i.c.study.controller.UserController    : 当前用户是user
   ，角色是user-role
   ```

5. 访问*http://localhost:8010/user-admin/1*，可以正常获得查询结果，同时可以看到用户微服务打印类似以下的日志：

   ```
   [nio-8000-exec-5] c.i.c.study.controller.UserController    : 当前用户是
       admin，角色是admin-role
   ```

由测试不难发现，手动创建 Feign 的方式更加灵活。

6.5 Feign 对继承的支持

Feign 支持继承。使用继承，可将一些公共操作分组到一些父接口中，从而简化 Feign 的开发，以下是个简单的例子。

基础接口：UserService.java

```
public interface UserService {
  @RequestMapping(method = RequestMethod.GET, value ="/users/{id}")
  User getUser(@PathVariable("id") long id);
}
```

服务提供者 Controller：UserResource.java

```
@RestController
public class UserResource implements UserService {
  //...
}
```

服务消费者：UserClient.java

```
@FeignClient("users")
public interface UserClient extends UserService {
}
```

> 尽管 Feign 的继承可帮助我们进一步简化 Feign 的开发，但 Spring Cloud 官方指出——通常情况下，**不建议在服务器端与客户端之间共享接口**，因为这种方式造成了服务器端与客户端代码的紧耦合。并且，Feign 本身并不使用 Spring MVC 的工作机制（方法参数映射不被继承）。

> 笔者认为，应客观看待“紧耦合”与“方便性”，并在权衡利弊后作出取舍——放弃开发的方便性或接受代码的紧耦合。以下几篇帖子对“紧耦合”与“方便性”进行了深入的讨论，有兴趣的读者可前往查看：
>
> - *https://github.com/spring-cloud/spring-cloud-netflix/issues/951*
> - *https://github.com/spring-cloud/spring-cloud-netflix/issues/659*
> - *https://github.com/spring-cloud/spring-cloud-netflix/issues/646*
> - *https://jmnarloch.wordpress.com/2015/08/19/spring-cloud-designing-feign-client/*

6.6 Feign 对压缩的支持

一些场景下，可能需要对请求或响应进行压缩，此时可使用以下属性启用 Feign 的压缩功能。

```
feign.compression.request.enabled=true
feign.compression.response.enabled=true
```

对于请求的压缩，Feign 还提供了更为详细的设置，例如：

```
feign.compression.request.enabled=true
feign.compression.request.mime-types=text/xml,application/xml,application/json
feign.compression.request.min-request-size=2048
```

其中，`feign.compression.request.mime-types`用于支持的媒体类型列表，默认是 text/xml、application/xml 以及 application/json。

`feign.compression.request.min-request-size`用于设置请求的最小阈值，默认是 2048。

> 该特性在 Spring Cloud Camden SR4 中并不生效。笔者已向官方反馈该问题，相信未来的版本中很快会被解决。详见：*https://github.com/spring-cloud/spring-cloud-netflix/issues/1580*。

6.7 Feign 的日志

很多场景下，需要了解 Feign 处理请求的具体细节，那么如何满足这种需求呢？

Feign 对日志的处理非常灵活，可为每个 Feign 客户端指定日志记录策略，每个 Feign 客户端都会创建一个 logger。默认情况下，logger 的名称是 Feign 接口的完整类名。需要注意的是，Feign 的日志打印只会对 DEBUG 级别做出响应。

我们可为每个 Feign 客户端配置各自的`Logger.Level`对象，告诉 Feign 记录哪些日志。Logger.Level 的值有以下选择。

- NONE：不记录任何日志（默认值）。
- BASIC：仅记录请求方法、URL、响应状态代码以及执行时间。
- HEADERS：记录 BASIC 级别的基础上，记录请求和响应的 header。
- FULL：记录请求和响应的 header，body 和元数据。

下面来为前面编写的`UserFeignClient`添加日志打印，将它的日志级别设置为 FULL。

1. 复制项目`microservice-consumer-movie-feign`，将 ArtifactId 修改为`microservice-consumer-movie-feign-logging`。
2. 编写 Feign 配置类：

```
@Configuration
public class FeignLogConfiguration {
  @Bean
  Logger.Level feignLoggerLevel() {
    return Logger.Level.FULL;
  }
}
```

3. 修改 Feign 的接口，指定配置类：

```
@FeignClient(name = "microservice-provider-user", configuration =
    FeignLogConfiguration.class)
public interface UserFeignClient {
  @RequestMapping(value = "/{id}", method = RequestMethod.GET)
  public User findById(@PathVariable("id") Long id);
}
```

4. 在 application.yml 中添加以下内容，指定 Feign 接口的日志级别为 DEBUG：

```
logging:
  level:
    com.itmuch.cloud.study.user.feign.UserFeignClient: DEBUG # 将Feign接口的日
      志级别设置成DEBUG，因为Feign的Logger.Level只对DEBUG作出响应。
```

测试

1. 启动 microservice-discovery-eureka。
2. 启动 microservice-provider-user。
3. 启动 microservice-consumer-movie-feign-logging。
4. 访问*http://localhost:8010/user/1*，可以看到类似于如下的日志。

```
2016-11-13 19:16:01.150 DEBUG 19200 --- [provider-user-3] c.i.c.study.user.
feign.UserFeignClient   : [UserFeignClient#findById] ---> GET http://micro
service-provider-user/1 HTTP/1.1
2016-11-13 19:16:01.150 DEBUG 19200 --- [provider-user-3] c.i.c.study.user.
feign.UserFeignClient   : [UserFeignClient#findById] ---> END HTTP (0-byte
body)
2016-11-13 19:16:01.164 DEBUG 19200 --- [provider-user-3] c.i.c.study.user.
feign.UserFeignClient   : [UserFeignClient#findById] <--- HTTP/1.1 200 (13
ms)
2016-11-13 19:16:01.165 DEBUG 19200 --- [provider-user-3] c.i.c.study.user.
feign.UserFeignClient   : [UserFeignClient#findById] content-type: applica
tion/json;charset=UTF-8
2016-11-13 19:16:01.165 DEBUG 19200 --- [provider-user-3] c.i.c.study.user.
feign.UserFeignClient   : [UserFeignClient#findById] date: Sun, 13 Nov 2016
11:16:01 GMT
2016-11-13 19:16:01.165 DEBUG 19200 --- [provider-user-3] c.i.c.study.user.
feign.UserFeignClient   : [UserFeignClient#findById] transfer-encoding:
```

```
chunked
2016-11-13 19:16:01.165 DEBUG 19200 --- [provider-user-3] c.i.c.study.user.
feign.UserFeignClient   : [UserFeignClient#findById] x-application-context
: microservice-provider-user:8000
2016-11-13 19:16:01.165 DEBUG 19200 --- [provider-user-3] c.i.c.study.user.
feign.UserFeignClient   : [UserFeignClient#findById]
2016-11-13 19:16:01.167 DEBUG 19200 --- [provider-user-3] c.i.c.study.user.
feign.UserFeignClient   : [UserFeignClient#findById] {"id":1,"username":
"account1","name":"张三","age":20,"balance":100.00}
2016-11-13 19:16:01.167 DEBUG 19200 --- [provider-user-3] c.i.c.study.user.
feign.UserFeignClient   : [UserFeignClient#findById] <--- END HTTP (72-byte
body)
```

可以看到，Feign 已将请求的各种细节非常详细地记录了下来。

5. 将日志 Feign 的日志级别改为 BASIC，重启项目后再次访问*http://localhost:8010/user/1*，日志如下：

```
2016-11-13 19:18:42.562 DEBUG 20176 --- [provider-user-4] c.i.c.study.user.
feign.UserFeignClient   : [UserFeignClient#findById] ---> GET http://micro
service-provider-user/1 HTTP/1.1
2016-11-13 19:18:42.582 DEBUG 20176 --- [provider-user-4] c.i.c.study.user.
feign.UserFeignClient   : [UserFeignClient#findById] <--- HTTP/1.1 200 (20
ms)
```

此时，只打印了请求方法、请求的 URL 和相应的状态码和响应的时间。

6.8 使用 Feign 构造多参数请求

本节探讨如何使用 Feign 构造多参数的请求，以 GET 以及 POST 方法的请求为例，其他方法（例如 DELETE、PUT 等）的请求原理相通，读者可自行研究。

6.8.1 GET 请求多参数的 URL

假设请求的 URL 包含多个参数，例如 *http://microservice-provider-user/get?id=1&username=*张三，要如何构造呢？

我们知道，Spring Cloud 为 Feign 添加了 Spring MVC 的注解支持，那么不妨按照 Spring MVC 的写法尝试一下：

```
@FeignClient("microservice-provider-user")
public interface UserFeignClient {
  @RequestMapping(value = "/get", method = RequestMethod.GET)
  public User get0(User user);
}
```

然而，这种写法并不正确，控制台会输出类似如下的异常。

```
feign.FeignException: status 405 reading UserFeignClient#get0(User); content:
{"timestamp":1482676142940,"status":405,"error":"Method Not Allowed","exception":"
    org.springframework.web.HttpRequestMethodNotSupportedException","message":"
    Request method 'POST' not supported","path":"/get"}
```

由异常可知，尽管指定了 GET 方法，Feign 依然会使用 POST 方法发送请求。

正确写法如下：

- 方法一

```
@FeignClient(name = "microservice-provider-user")
public interface UserFeignClient {
  @RequestMapping(value = "/get", method = RequestMethod.GET)
  public User get1(@RequestParam("id") Long id, @RequestParam("username") String
      username);
}
```

这是最为直观的方式，URL 有几个参数，Feign 接口中的方法就有几个参数。使用@RequestParam注解指定请求的参数是什么。

- 方法二

多参数的 URL 也可使用 Map 来构建。当目标 URL 参数非常多时，可使用这种方式简化 Feign 接口的编写。

```
@FeignClient(name = "microservice-provider-user")
public interface UserFeignClient {
  @RequestMapping(value = "/get", method = RequestMethod.GET)
  public User get2(@RequestParam Map<String, Object> map);
}
```

在调用时，可使用类似以下的代码。

```
public User get(String username, String password) {
  HashMap<String, Object> map = Maps.newHashMap();
  map.put("id", "1");
```

```
  map.put("username", "张三");
  return this.userFeignClient.get2(map);
}
```

6.8.2 POST 请求包含多个参数

下面来讨论如何使用 Feign 构造包含多个参数的 POST 请求。假设服务提供者的 Controller 是这样编写的：

```
@RestController
public class UserController {
  @PostMapping("/post")
  public User post(@RequestBody User user) {
    ...
  }
}
```

要如何使用 Feign 去请求呢？答案非常简单，示例：

```
@FeignClient(name = "microservice-provider-user")
public interface UserFeignClient {
  @RequestMapping(value = "/post", method = RequestMethod.POST)
  public User post(@RequestBody User user);
}
```

- 本节相关代码，详见本书配套代码中的 microservice-provider-user-multiple-params 项目和 microservice-consumer-movie-feign-multiple-params 项目。
- 除本节讲解的方式外，还可编写自己的编码器来构造多参数的请求，但这种方式编码成本较高，代码可重用性较低。故此，本书不再赘述。

7 使用 Hystrix 实现微服务的容错处理

至此，已用 Eureka 实现了微服务的注册与发现，Ribbon 实现了客户端侧的负载均衡，Feign 实现了声明式的 API 调用。

本章讨论如何使用 Hystrix 实现微服务的容错。

7.1 实现容错的手段

如果服务提供者响应非常缓慢，那么消费者对提供者的请求就会被强制等待，直到提供者响应或超时。在高负载场景下，如果不作任何处理，此类问题可能会导致服务消费者的资源耗竭甚至整个系统的崩溃。例如，曾经发生过一个案例——某电子商务网站在一个黑色星期五发生过载。过多的并发请求，导致用户支付的请求延迟很久都没有响应，在等待很长时间后最终失败。支付失败又导致用户重新刷新页面并再次尝试支付，进一步增加了服务器的负载，最终整个系统都崩溃了。

当依赖的服务不可用时，服务自身会不会被拖垮？这是我们要考虑的问题。

7.1.1 雪崩效应

微服务架构的应用系统通常包含多个服务层。微服务之间通过网络进行通信，从而支撑起整个应用系统，因此，微服务之间难免存在依赖关系。我们知道，任何微服务都并非 100% 可用，网络往往也很脆弱，因此难免有些请求会失败。

我们常把“基础服务故障”导致“级联故障”的现象称为雪崩效应。雪崩效应描述的是提供者不可用导致消费者不可用，并将不可用逐渐放大的过程。

如图 7-1，A 作为服务提供者（基础服务），B 为 A 的服务消费者，C 和 D 是 B 的服务消费者。当 A 不可用引起了 B 的不可用，并将不可用像滚雪球一样放大到 C 和 D 时，雪崩效应就形成了。

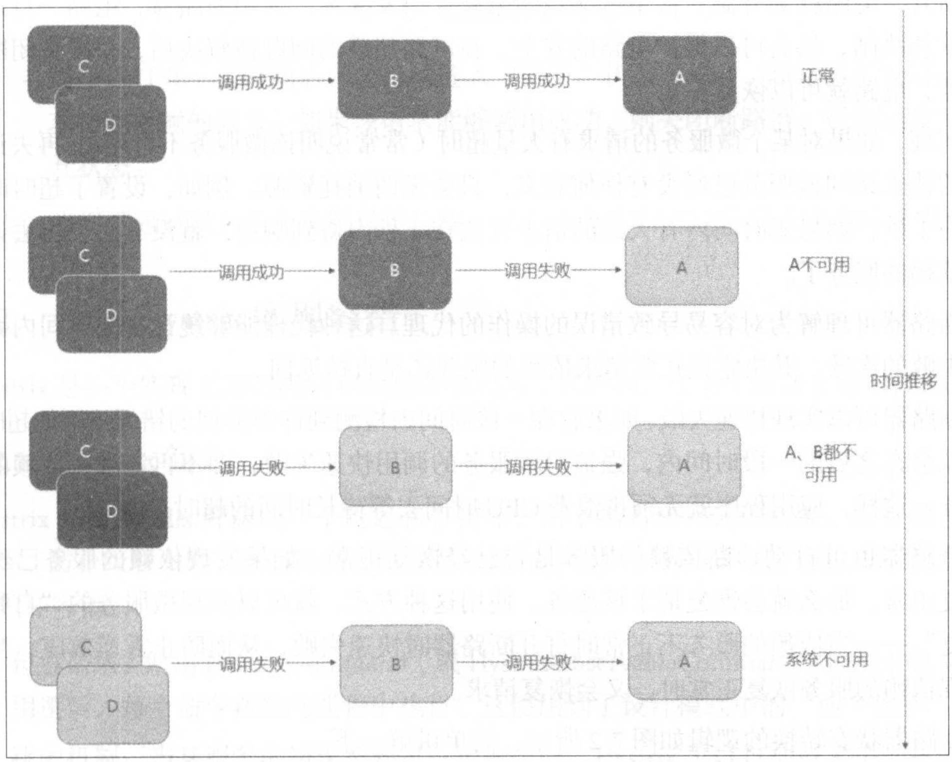


图 7-1 雪崩效应形成过程

7.1.2 如何容错

要想防止雪崩效应，必须有一个强大的容错机制。该容错机制需实现以下两点。

- 为网络请求设置超时

  必须为网络请求设置超时。正常情况下，一个远程调用一般在几十毫秒内就能得到响应了。如果依赖的服务不可用或者网络有问题，那么响应时间就会变得很长（几十秒）。

  通常情况下，一次远程调用对应着一个线程/进程。如果响应太慢，这个线程/进程就得不到释放。而线程/进程又对应着系统资源，如果得不到释放的线程/进程越积越多，资源就会逐渐被耗尽，最终导致服务的不可用。

  因此，必须为每个网络请求设置超时，让资源尽快释放。

- 使用断路器模式

  试想一下，如果家里没有断路器，当电流过载时（例如功率过大、短路等），电路不断开，电路就会升温，甚至可能烧断电路、引发火灾。使用断路器，电路一旦过载就会跳闸，从而可以保护电路的安全。在电路超载的问题被解决后，只须关闭断路器，电路就可以恢复正常。

  同理，如果对某个微服务的请求有大量超时（常常说明该微服务不可用），再去让新的请求访问该服务已经没有任何意义，只会无谓消耗资源。例如，设置了超时时间为 1 秒，如果短时间内有大量的请求无法在 1 秒内得到响应，就没有必要再去请求依赖的服务了。

  断路器可理解为对容易导致错误的操作的代理。这种代理能够统计一段时间内调用失败的次数，并决定是正常请求依赖的服务还是直接返回。

  断路器可以实现快速失败，如果它在一段时间内检测到许多类似的错误（例如超时），就会在之后的一段时间内，强迫对该服务的调用快速失败，即不再请求所依赖的服务。这样，应用程序就无须再浪费 CPU 时间去等待长时间的超时。

  断路器也可自动诊断依赖的服务是否已经恢复正常。如果发现依赖的服务已经恢复正常，那么就会恢复请求该服务。使用这种方式，就可以实现微服务的“自我修复”——当依赖的服务不正常时打开断路器时快速失败，从而防止雪崩效应；当发现依赖的服务恢复正常时，又会恢复请求。

  断路器状态转换的逻辑如图 7-2 所示，简单讲解一下。

  - 正常情况下，断路器关闭，可正常请求依赖的服务。
  - 当一段时间内，请求失败率达到一定阈值（例如错误率达到 50%，或 100 次/分钟等），断路器就会打开。此时，不会再去请求依赖的服务。

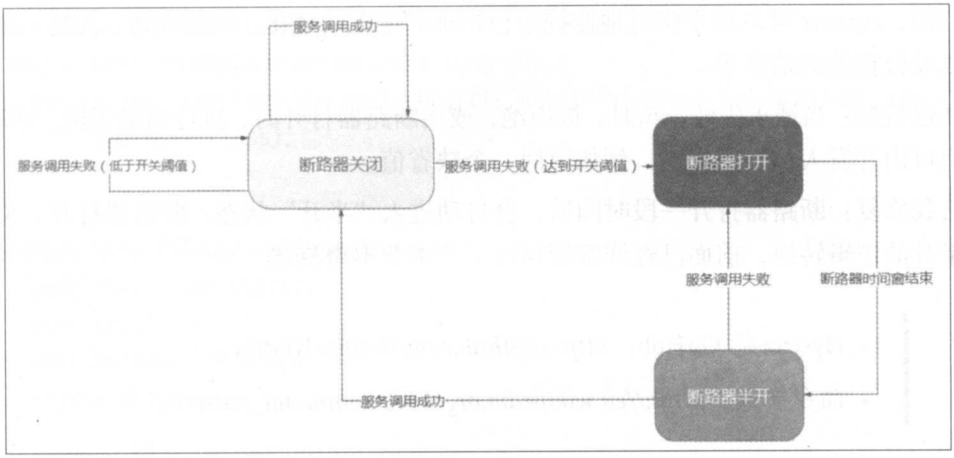


图 7-2　断路器状态转换图

- 断路器打开一段时间后，会自动进入“半开”状态。此时，断路器可允许一个请求访问依赖的服务。如果该请求能够调用成功，则关闭断路器；否则继续保持打开状态。

综上，我们可通过以上两点机制保护应用，从而防止雪崩效应并提升应用的可用性。

7.2　使用 Hystrix 实现容错

Hystrix 是一个实现了超时机制和断路器模式的工具类库。先来了解什么是 Hystrix。

7.2.1　Hystrix 简介

Hystrix 是由 Netflix 开源的一个延迟和容错库，用于隔离访问远程系统、服务或者第三方库，防止级联失败，从而提升系统的可用性与容错性。Hystrix 主要通过以下几点实现延迟和容错。

- 包裹请求：使用 HystrixCommand（或 HystrixObservableCommand）包裹对依赖的调用逻辑，每个命令在独立线程中执行。这使用到了设计模式中的“命令模式”。
- 跳闸机制：当某服务的错误率超过一定阈值时，Hystrix 可以自动或者手动跳闸，停止请求该服务一段时间。
- 资源隔离：Hystrix 为每个依赖都维护了一个小型的线程池（或者信号量）。如果该线程池已满，发往该依赖的请求就被立即拒绝，而不是排队等候，从而加速失败判定。

- 监控：Hystrix 可以近乎实时地监控运行指标和配置的变化，例如成功、失败、超时、以及被拒绝的请求等。
- 回退机制：当请求失败、超时、被拒绝，或当断路器打开时，执行回退逻辑。回退逻辑可由开发人员自行提供，例如返回一个缺省值。
- 自我修复：断路器打开一段时间后，会自动进入“半开”状态。断路器打开、关闭、半开的逻辑转换，前面已经详细探讨过了，本节不再赘述。

> - Hystrix 的 GitHub：*https://github.com/Netflix/Hystrix*。
> - 命令模式：*https://en.wikipedia.org/wiki/Command_pattern*。

7.2.2 通用方式整合 Hystrix

在 Spring Cloud 中，整合 Hystrix 非常方便。以项目`microservice-consumer-movie-ribbon`为例，我们来为它整合 Hystrix。

1. 复制项目`microservice-consumer-movie-ribbon`，将 ArtifactId 修改为 `microservice-consumer-movie-ribbon-hystrix`。
2. 为项目添加以下依赖。

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-hystrix</artifactId>
</dependency>
```

3. 在启动类上添加注解`@EnableCircuitBreaker`或`@EnableHystrix`，从而为项目启用断路器支持。
4. 修改 MovieController，让其中的 findById 方法具备容错能力。

```
@RestController
public class MovieController {
  private static final Logger LOGGER = LoggerFactory.getLogger(MovieController.
      class);
  @Autowired
  private RestTemplate restTemplate;
  @Autowired
  private LoadBalancerClient loadBalancerClient;

  @HystrixCommand(fallbackMethod = "findByIdFallback")
```

```
  @GetMapping("/user/{id}")
  public User findById(@PathVariable Long id) {
    return this.restTemplate.getForObject("http://microservice-provider-user/" +
         id, User.class);
  }

  public User findByIdFallback(Long id) {
    User user = new User();
    user.setId(-1L);
    user.setName("默认用户");
    return user;
  }
  ...
}
```

由代码可知，为 findById 方法编写了一个回退方法 findByIdFallback，该方法与 findById 方法具有相同的参数与返回值类型，该方法返回了一个默认的 User。

在 findById 方法上，使用注解`@HystrixCommand`的 fallbackMethod 属性，指定回退方法是 findByIdFallback。注解`@HystrixCommand`由名为 javanica（*https://github.com/Netflix/Hystrix/tree/master/hystrix-contrib/hystrix-javanica*）的 Hystrix contrib 库提供。javanica 是一个 Hystrix 的子项目，用于简化 Hystrix 的使用。Spring Cloud 自动将 Spring bean 与该注解封装在一个连接到 Hystrix 断路器的代理中。

`@HystrixCommand`的配置非常灵活，可使用注解`@HystrixProperty`的 commandProperties 属性来配置`@HystrixCommand`。例如：

```
@HystrixCommand(fallbackMethod = "findByIdFallback", commandProperties = {
    @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds",
        value = "5000"),
    @HystrixProperty(name = "metrics.rollingStats.timeInMilliseconds", value = "
        10000")
}, threadPoolProperties = {
    @HystrixProperty(name = "coreSize", value = "1"),
    @HystrixProperty(name = "maxQueueSize", value = "10")
})
@GetMapping("/user/{id}")
public User findById(@PathVariable Long id) {
  // ...
}
```

限于篇幅，笔者就不一一讲解 Hystrix 的配置属性了。

读者可前往*https://github.com/Netflix/Hystrix/tree/master/hystrix-contrib/hystrix-javanica#configuration* 详细了解注解`@HystrixCommand`如何使用。

Hystrix 配置属性可详见 Hystrix Wiki：*https://github.com/Netflix/Hystrix/wiki/Configuration*。

测试

1. 启动项目 microservice-discovery-eureka。
2. 启动项目 microservice-provider-user。
3. 启动项目 microservice-consumer-movie-ribbon-hystrix。
4. 访问*http://localhost:8010/user/1*，可获得如下的结果。

```
{
  "id": 1,
  "username": "account1",
  "name": "张三",
  "age": 20,
  "balance": 100
}
```

5. 停止 microservice-provider-user。
6. 再次访问*http://localhost:8010/user/1*，获得如下结果。

```
{
  "id": -1,
  "username": null,
  "name": "默认用户",
  "age": null,
  "balance": null
}
```

说明当用户微服务不可用时，进入了回退方法。

我们知道，当请求失败、被拒绝、超时或者断路器打开时，都会进入回退方法。但进入回退方法并不意味着断路器已经被打开。那么，如何才能明确了解断路器当前的状态呢？请听下回分解。

7.2.3 Hystrix 断路器的状态监控与深入理解

还记得之前为项目引入了 Spring Boot Actuator 吗？

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

断路器的状态也会暴露在 Actuator 提供的/health 端点中，这样就可以直观地了解断路器的状态。下面来做一点实验，深入理解断路器的状态转换。

测试

1. 启动项目 microservice-discovery-eureka。
2. 启动项目 microservice-provider-user。
3. 启动项目 microservice-consumer-movie-ribbon-hystrix。
4. 访问*http://localhost:8010/user/1*，可正常获得结果。
5. 访问*http://localhost:8010/health*，可获得类似如下结果。

```
{
  "status": "UP",
  "hystrix": {
    "status": "UP"
  }
  ...
}
```

   可以看到此时，Hystrix 的状态是 UP，也就是一切正常，此时断路器是关闭的。
6. 停止 microservice-provider-user，访问*http://localhost:8010/user/1*，可获得如下结果。

```
{
  "id": -1,
  "username": null,
  "name": "默认用户",
  "age": null,
  "balance": null
}
```

7. 访问*http://localhost:8010/health*，可获得如下结果。

```
{
  "status": "UP",
  "hystrix": {
      "status": "UP"
  }
  ...
}
```

我们发现，尽管执行了回退逻辑，返回了默认用户，但此时 Hystrix 的状态依然是 UP，这是因为我们的失败率还没有达到阈值（默认是 5 秒内 20 次失败）。这是很多初学者会遇到的误区，这边**再次强调**——执行回退逻辑并不代表断路器已经打开。请求失败、超时、被拒绝以及断路器打开时等都会执行回退逻辑。

8. 持续快速地访问*http://localhost:8010/user/1*，直到请求快速返回。

```
{
  "status": "UP",
  "hystrix": {
    "status": "CIRCUIT_OPEN",
    "openCircuitBreakers": [
       "MovieController::findById"
    ]
  }
  ...
}
```

可以看到，Hystrix 的状态是 CIRCUIT_OPEN，说明断路器已经打开，不会再去请求用户微服务了。我们也可使用类似的方法测试断路器从打开转半开以及从半开自动恢复等过程。

7.2.4 Hystrix 线程隔离策略与传播上下文

本节相对复杂，为了讲解这个问题，先来阅读一下 Hystrix 官方 Wiki（*https://github.com/Netflix/Hystrix/wiki/Configuration#execution.isolation.strategy*）。

execution.isolation.strategy

This property indicates which isolation strategy `HystrixCommand.run()` executes with, one of the following two choices:

- `THREAD` —it executes on a separate thread and concurrent requests are limited by the number of threads in the thread-pool

> - SEMAPHORE —it executes on the calling thread and concurrent requests are limited by the semaphore count
>
> **Thread or Semaphore**
>
> The default, and the recommended setting, is to run commands using thread isolation (THREAD).
>
> Commands executed in threads have an extra layer of protection against latencies beyond what network timeouts can offer.
>
> Generally the only time you should use semaphore isolation (SEMAPHORE) is when the call is so high volume (hundreds per second, per instance) that the overhead of separate threads is too high; this typically only applies to non-network calls.

简单翻译一下，Hystrix 的隔离策略有两种：分别是线程隔离和信号量隔离。

- THREAD（线程隔离）：使用该方式，HystrixCommand 将会在单独的线程上执行，并发请求受线程池中的线程数量的限制。
- SEMAPHORE（信号量隔离）：使用该方式，HystrixCommand 将会在调用线程上执行，开销相对较小，并发请求受到信号量个数的限制。

Hystrix 中默认并且推荐使用线程隔离（THREAD），因为这种方式有一个除网络超时以外的额外保护层。

一般来说，只有当调用负载非常高时（例如每个实例每秒调用数百次）才需要使用信号量隔离，因为这种场景下使用 THREAD 开销会比较高。信号量隔离一般仅适用于非网络调用的隔离。

可使用`execution.isolation.strategy`属性指定隔离策略。

了解 Hystrix 的隔离策略后，再来看一下 Spring Cloud 官方的文档：

> If you want some thread local context to propagate into a @HystrixCommand the default declaration will not work because it executes the command in a thread pool (in case of timeouts). You can switch Hystrix to use the same thread as the caller using some configuration, or directly in the annotation, by asking it to use a different "Isolation Strategy". For example:

```
@HystrixCommand(fallbackMethod = "stubMyService",
    commandProperties = {
```

```
        @HystrixProperty(name="execution.isolation.strategy", value="
            SEMAPHORE")
    }
)
...
```

The same thing applies if you are using `@SessionScope` or `@RequestScope`. You will know when you need to do this because of a runtime exception that says it can't find the scoped context.

You also have the option to set the `hystrix.shareSecurityContext` property to `true`. Doing so will auto configure an Hystrix concurrency strategy plugin hook who will transfer the `SecurityContext` from your main thread to the one used by the Hystrix command. Hystrix does not allow multiple hystrix concurrency strategy to be registered so an extension mechanism is available by declaring your own `HystrixConcurrencyStrategy` as a Spring bean. Spring Cloud will lookup for your implementation within the Spring context and wrap it inside its own plugin.

简单翻译一下：

如果你想传播线程本地的上下文到`@HystrixCommand`，默认声明将不会工作，因为它会在线程池中执行命令（在超时的情况下）。你可以使用一些配置，让 Hystrix 使用相同的线程，或者直接在注解中让 Hystrix 使用不同的隔离策略。例如：

```
@HystrixCommand(fallbackMethod = "stubMyService",
    commandProperties = {
      @HystrixProperty(name="execution.isolation.strategy", value="SEMAPHORE")
    }
)
```

这也适用于使用`@SessionScope`或者`@RequestSession`的情况。你会知道什么时候需要这样做，因为会发生一个运行时异常，说它找不到作用域上下文（scoped context）。

你还可将`hystrix.shareSecurityContext`属性设置为 true，这样将会自动配置一个 Hystrix 并发策略插件的 hook，这个 hook 会将 SecurityContext 从主线程传输到 Hystrix 的命令。因为 Hystrix 不允许注册多个 Hystrix 并发策略，所以可以声明 HystrixConcurrencyStrategy 为一个 Spring bean 来实现扩展。Spring Cloud 会在 Spring 的上下文中查找你的实现，并将其包装在自己的插件中。

总结

把 Spring Cloud 和 Hystrix 的文档对照阅读，就能很好地理解相关概念。在此，笔者总结如下：

- Hystrix 的隔离策略有 THREAD 和 SEMAPHORE 两种，默认是 THREAD。
- 正常情况下，保持默认即可。
- 如果发生找不到上下文的运行时异常，可考虑将隔离策略设置为 SEMAPHORE。

> - 在 Github 上的相关 issue，可帮助大家理解：*https://github.com/spring-cloud/spring-cloud-netflix/issues/1336*。
> - 服务熔断、降级、限流、异步 RPC -- HyStrix：*http://blog.csdn.net/chunlongyu/article/details/53259014*。
> - 分布式系统延迟和容错框架 Hystrix：*http://blog.csdn.net/fight4gold/article/details/51252217*。

7.2.5 Feign 使用 Hystrix

前文是使用注解`@HystrixCommand`的 fallbackMethod 属性实现回退的。然而，Feign 是以接口形式工作的，它没有方法体，前文讲解的方式显然不适用于 Feign。

那么 Feign 要如何整合 Hystrix 呢？不仅如此，如何实现 Feign 的回退呢？

事实上，Spring Cloud 默认已为 Feign 整合了 Hystrix，只要 Hystrix 在项目的 classpath 中，Feign 默认就会用断路器包裹所有方法。下面来详细探讨如何实现 Feign 的回退。

7.2.5.1 为 Feign 添加回退

本节为前文编写的 UserFeignClient 添加回退。

1. 复制项目`microservice-consumer-movie-feign`，将 ArtifactId 修改为 `microservice-consumer-movie-feign-hystrix-fallback`。
2. 将之前编写的 Feign 接口修改成如下内容：

```
/**
 * Feign的fallback测试
 * 使用@FeignClient的fallback属性指定回退类
 * @author 周立
 */
```

```
@FeignClient(name = "microservice-provider-user", fallback = FeignClientFallback
    .class)
public interface UserFeignClient {
  @RequestMapping(value = "/{id}", method = RequestMethod.GET)
  public User findById(@PathVariable("id") Long id);

}

/**
 * 回退类FeignClientFallback需实现Feign Client接口
 * FeignClientFallback也可以是public class，没有区别
 * @author 周立
 */
@Component
class FeignClientFallback implements UserFeignClient {
  @Override
  public User findById(Long id) {
    User user = new User();
    user.setId(-1L);
    user.setUsername("默认用户");
    return user;
  }
}
```

由代码可知，只须使用`@FeignClient`注解的 fallback 属性，就可为指定名称的 Feign 客户端添加回退。

测试

1. 启动 microservice-discovery-eureka。
2. 启动 microservice-provider-user。
3. 启动 microservice-consumer-movie-feign-hystrix-fallback。
4. 访问*http://localhost:8010/user/1*，可正常获得结果。
5. 停止 microservice-provider-user。
6. 再次访问*http://localhost:8010/user/1*，可获得如下结果。说明当用户微服务不可用时，进入了回退的逻辑。

```
  {
    "id": -1,
    "username": "默认用户",
    "name": null,
    "age": null,
    "balance": null
  }
```

7.2.5.2 通过 Fallback Factory 检查回退原因

很多场景下，需要了解回退的原因，此时可使用注解`@FeignClient`的 fallbackFactory 属性。下面来编写一个示例，为 Feign 打印回退日志。

1. 复制项目`microservice-consumer-movie-feign`，将 ArtifactId 修改为 `microservice-consumer-movie-feign-hystrix-fallback-factory`。
2. 将 UserFeignClient 改为如下内容。

```
@FeignClient(name = "microservice-provider-user", fallbackFactory =
    FeignClientFallbackFactory.class)
public interface UserFeignClient {
  @RequestMapping(value = "/{id}", method = RequestMethod.GET)
  public User findById(@PathVariable("id") Long id);
}

/**
 * UserFeignClient的fallbackFactory类，该类需实现FallbackFactory接口，并覆写
   create方法
 * The fallback factory must produce instances of fallback classes that
 * implement the interface annotated by {@link FeignClient}.
 * @author 周立
 */
@Component
class FeignClientFallbackFactory implements FallbackFactory<UserFeignClient> {
  private static final Logger LOGGER = LoggerFactory.getLogger(
      FeignClientFallbackFactory.class);

  @Override
  public UserFeignClient create(Throwable cause) {
    return new UserFeignClient() {
```

```
      @Override
      public User findById(Long id) {
        // 日志最好放在各个fallback方法中，而不要直接放在create方法中。
        // 否则在引用启动时，就会打印该日志。
        // 详见https://github.com/spring-cloud/spring-cloud-netflix/issues/1471
        FeignClientFallbackFactory.LOGGER.info("fallback; reason was:", cause);
        User user = new User();
        user.setId(-1L);
        user.setUsername("默认用户");
        return user;
      }
    };
  }
}
```

这样，当 Feign 发生回退时，就会打印日志。

测试

1. 启动 microservice-discovery-eureka。
2. 启动 microservice-provider-user。
3. 启动 microservice-consumer-movie-feign-hystrix-fallback-factory。
4. 访问*http://localhost:8010/user/1* 能正常获得结果。
5. 停止 microservice-provider-user。
6. 再次访问*http://localhost:8010/user/1*，可获得如下结果。

```
{
  "id": -1,
  "username": "默认用户",
  "name": null,
  "age": null,
  "balance": null
}
```

并且，控制台会输出类似如下的日志。

```
INFO 23296 --- [provider-user-1] c.i.c.s.u.f.FeignClientFallbackFactory   :
fallback; reason was: com.netflix.client.ClientException: Load balancer does
not have available server for client: microservice-provider-user
```

说明进入了回退类中的回退方法。

1. fallbackFactory 属性还有很多其他的用途，例如让不同的异常返回不同的回退结果，从而使 Feign 的回退更加灵活。例如：

```
@Component
class FeignClientFallbackFactory implements FallbackFactory<
    UserFeignClient> {
  @Override
  public UserFeignClient create(Throwable cause) {
    return new UserFeignClient() {
      @Override
      public User findById(Long id) {
        User user = new User();
        if (cause instanceof IllegalArgumentException) {
          user.setId(-1L);
        } else {
          user.setId(-2L);
        }
        return user;
      }
    };
  }
}
```

2. 在特定的时间窗口中，create(Throwable cause) 中的 cause 可能是 null。这是 Feign 的 Bug，该 Bug 在 Feign 9.4.0 中已被解决。由于 Spring Cloud Camden SR4 使用的 Feign 版本是 9.3.1，因此，在使用该特性时，如需访问 cause 变量中的属性，需添加一些判断代码，从而规避空指针异常。对该 Bug 感兴趣的朋友可详见该 Issue：*https://github.com/OpenFeign/feign/issues/464*。

7.2.5.3 为 Feign 禁用 Hystrix

前文说过，在 Spring Cloud 中，只要 Hystrix 在项目的 classpath 中，Feign 就会使用断路器包裹 Feign 客户端的所有方法。这样虽然方便，但很多场景下并不需要该功能。如何为 Feign 禁用 Hystrix 呢？

为指定 Feign 客户端禁用 Hystrix

借助 Feign 的自定义配置，可轻松为指定名称的 Feign 客户端禁用 Hystrix。例如：

```
@Configuration
public class FeignDisableHystrixConfiguration {
    @Bean
    @Scope("prototype")
    public Feign.Builder feignBuilder() {
        return Feign.builder();
    }
}
```

想要禁用 Hystrix 的`@FeignClient`引用该配置类即可，例如：

```
@FeignClient(name = "user", configuration = FeignDisableHystrixConfiguration.class)
public interface UserFeignClient {
  //...
}
```

全局禁用 Hystrix

也可为 Feign 全局禁用 Hystrix。只须在 application.yml 中配置`feign.hystrix.enabled = false`即可。

7.3 Hystrix 的监控

除实现容错外，Hystrix 还提供了近乎实时的监控。HystrixCommand 和 HystrixObservableCommand 在执行时，会生成执行结果和运行指标，比如每秒执行的请求数、成功数等，这些监控数据对分析应用系统的状态很有用。

使用 Hystrix 的模块`hystrix-metrics-event-stream`，就可将这些监控的指标信息以`text/event-stream`的格式暴露给外部系统。`spring-cloud-starter-hystrix`已包含该模块，在此基础上，只须为项目添加`spring-boot-starter-actuator`，就可使用/hystrix.stream 端点获得 Hystrix 的监控信息了。

如上所述，前文的项目`microservice-consumer-movie-ribbon-hystrix`已具备监控 Hystrix 的能力。下面来做一点测试。

1. 启动 microservice-discovery-eureka。
2. 启动 microservice-provider-user。
3. 启动 microservice-consumer-movie-ribbon-hystrix。
4. 访问*http://localhost:8010/hystrix.stream* ，可看到浏览器一直处于请求的状态，页面空白。这是因为此时项目中注解了`@HystrixCommand`的方法还没有被执行，因此也没有

任何的监控数据。

5. 访问*http://localhost:8010/user/1*后，再次访问*http://localhost:8010/hystrix.stream*，可看到页面会重复出现类似于以下的内容。

```
data: {"type":"HystrixCommand","name":"findById","group":"MovieController","
    currentTime":1479303396533,"isCircuitBreakerOpen":false,"errorPercentage
    ":0,"errorCount":0,"requestCount":0,"rollingCountBadRequests":0,"
    rollingCountCollapsedRequests":0,"rollingCountEmit":0,"
    rollingCountExceptionsThrown":0,"rollingCountFailure":0,"
    rollingCountFallbackEmit":0,"rollingCountFallbackFailure":0,"
    rollingCountFallbackMissing":0,"rollingCountFallbackRejection":0,"
    rollingCountFallbackSuccess":0,"rollingCountResponsesFromCache":0,"
    rollingCountSemaphoreRejected":0,"rollingCountShortCircuited":0,"
    rollingCountSuccess":0,"rollingCountThreadPoolRejected":0,"
    rollingCountTimeout":0,"currentConcurrentExecutionCount":1,"
    rollingMaxConcurrentExecutionCount":0,"latencyExecute_mean":0,"
    latencyExecute":...}
```

这是因为系统会不断地刷新以获得实时的监控数据。Hystrix 的监控指标非常全面，例如 HystrixCommand 的名称、group 名称、断路器状态、错误率、错误数等。

Feign 项目的 Hystrix 监控

启动前文的`microservice-consumer-movie-feign-hystrix-fallback`项目，并使用类似的方式测试，然后访问*http://localhost:8010/hystrix.stream*，发现返回的是 404。这是为什么呢？

查看项目的依赖树会发现，项目甚至连`hystrix-metrics-event-stream`的依赖都没有。那么如何解决该问题呢？

解决方案如下：

1. 复制项目`microservice-consumer-movie-feign-hystrix-fallback`，将 ArtifactId 修改为`microservice-consumer-movie-feign-hystrix-fallback-stream`。
2. 为项目添加`spring-cloud-starter-hystrix`的依赖。

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-hystrix</artifactId>
</dependency>
```

3. 在启动类上添加`@EnableCircuitBreaker`，这样就可使用/hystrix.stream 端点监控 Hystrix 了。

7.4 使用 Hystrix Dashboard 可视化监控数据

前面讨论了 Hystrix 的监控，但访问/hystrix.stream 端点获得的数据是以文字形式展示的。很难通过这些数据，一眼看出系统当前的运行状态。

可使用 Hystrix Dashboard，从让监控数据图形化、可视化。

下面来编写一个 Hystrix Dashboard。

1. 创建一个 Maven 项目，ArtifactId 是microservice-hystrix-dashboard，并为项目添加以下依赖。

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>
</dependency>
```

2. 编写启动类，在启动类上添加@EnableHystrixDashboard。

```
@SpringBootApplication
@EnableHystrixDashboard
public class HystrixDashboardApplication {
  public static void main(String[] args) {
    SpringApplication.run(HystrixDashboardApplication.class, args);
  }
}
```

3. 在配置文件 application.yml 中添加如下内容。

```
server:
  port: 8030
```

这样，一个简单的 Hystrix Dashboard 就完成了。由配置可知，我们并没有把 Hystrix Dashboard 注册到 Eureka Server 上。

测试

1. 访问*http://localhost:8030/hystrix*，可看到 Hystrix Dashboard 的主页，如图 7-3所示。

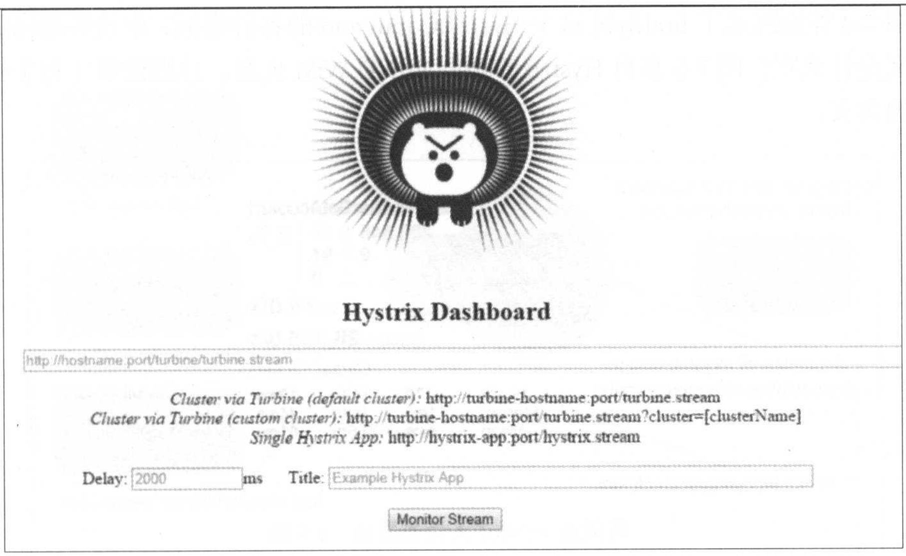


图 7-3 Hystrix Dashboard 主页

2. 在上一节测试的基础上，在 URL 一栏输入*http://localhost:8010/hystrix.stream*，随意设置一个 Title，并点击 Monitor Stream 按钮后，即可看到类似图 7-4 的界面。

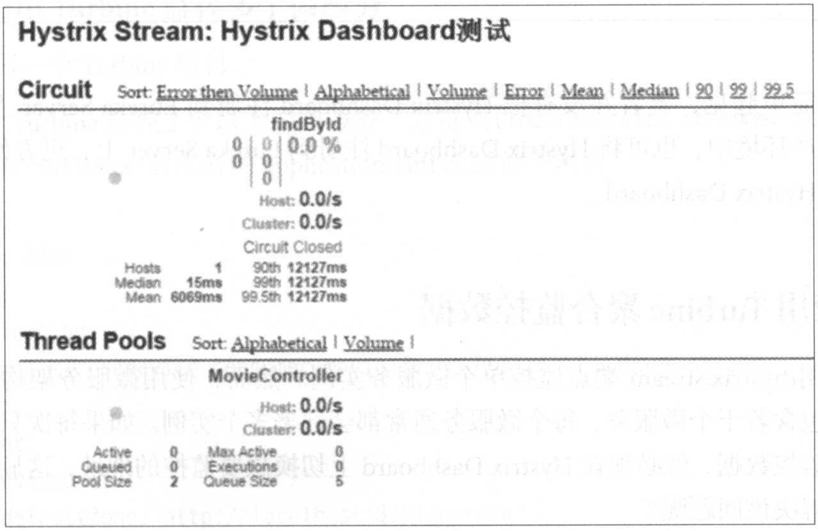


图 7-4 Hystrix Dashboard 监控页面

图 7-4 详细展示了 findById 这个 HystrixCommand 的各种指标，但这些指标的含义是什么呢？图 7-5 来自 Hystrix Dashboard 的 Wiki 页面，详细说明了每个指标的含义。

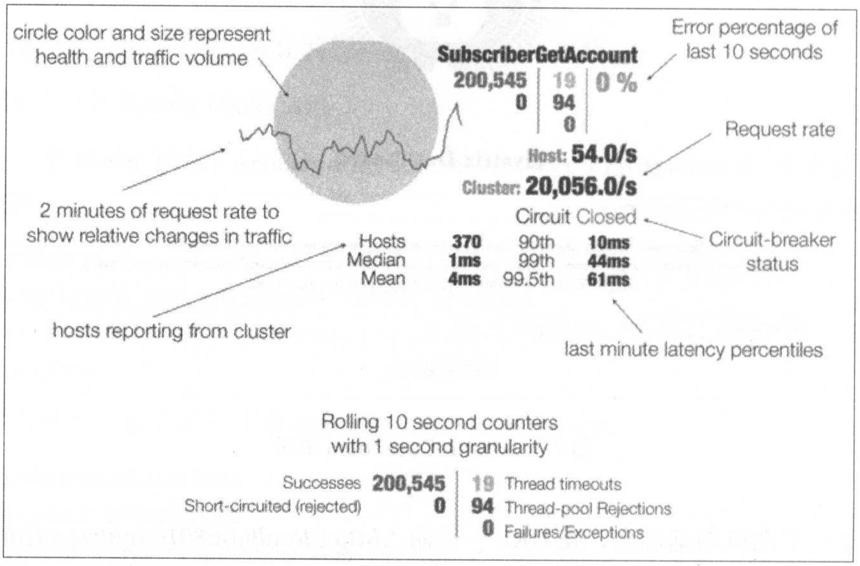


图 7-5 Hystrix Dashboar 指标解释

> 简单起见，笔者并没有把 Hystrix Dashboard 注册到 Eureka Server 上。在生产环境中，也可将 Hystrix Dashboard 注册到 Eureka Server 上，更方便地管理 Hystrix Dashboard。

7.5 使用 Turbine 聚合监控数据

前文中使用/hystrix.stream 端点监控单个微服务实例。然而，使用微服务架构的应用系统一般会包含若干个微服务，每个微服务通常都会部署多个实例。如果每次只能查看单个实例的监控数据，就必须在 Hystrix Dashboard 上切换想要监控的地址，这显然很不方便。如何解决该问题呢？

7.5.1 Turbine 简介

Turbine 是一个聚合 Hystrix 监控数据的工具，它可将所有相关/hystrix.stream 端点的数据聚合到一个组合的/turbine.stream 中，从而让集群的监控更加方便。

引入 Turbine 后，架构如图 7-6 所示。

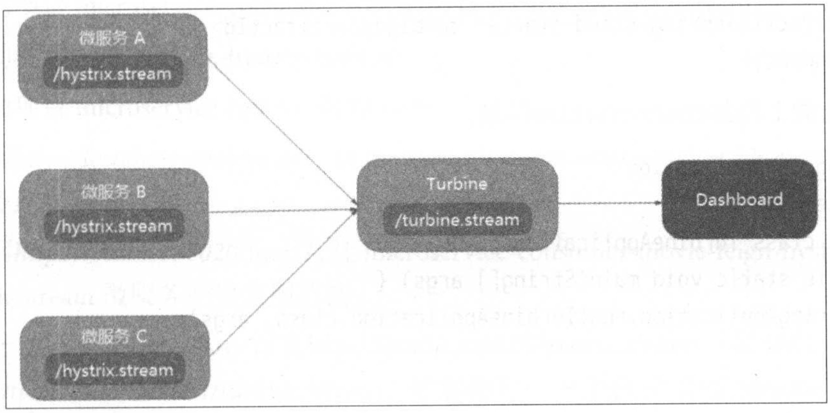


图 7-6 微服务引入 Turbine 架构图

> Turbine 的 GitHub：*https://github.com/Netflix/Turbine*。

7.5.2 使用 Turbine 监控多个微服务

本节来编写一个 Turbine 项目。

1. 为了让 Turbine 监控 2 个以上的微服务，先将项目`microservice-consumer-movie-feign-hystrix-fallback-stream`的 application.yml 改成如下内容：

```
server:
  port: 8020
spring:
  application:
    name: microservice-consumer-movie-feign-hystrix-fallback-stream
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
  instance:
    prefer-ip-address: true
```

2. 创建一个 Maven 项目，ArtifactId 是`microservice-hystrix-turbine`，并为项目添加以下依赖。

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-turbine</artifactId>
</dependency>
```

3. 在启动类上添加`@EnableTurbine`注解。

```
@SpringBootApplication
@EnableTurbine
public class TurbineApplication {
  public static void main(String[] args) {
    SpringApplication.run(TurbineApplication.class, args);
  }
}
```

4. 编写配置文件 application.yml。

```
server:
  port: 8031
spring:
  application:
    name: microservice-hystrix-turbine
eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/
  instance:
    prefer-ip-address: true
turbine:
  appConfig: microservice-consumer-movie,microservice-consumer-movie-feign-
      hystrix-fallback-stream
  clusterNameExpression: "'default'"
```

使用以上配置，Turbine 会在 Eureka Server 中找到`microservice-consumer-movie`和 `microservice-consumer-movie-feign-hystrix-fallback-stream` 这两个微服务，并聚合两个微服务的监控数据。

测试

1. 启动项目 microservice-discovery-eureka。
2. 启动项目 microservice-provider-user。

3. 启动项目 microservice-consumer-movie-ribbon-hystrix。
4. 启动项目 microservice-consumer-movie-feign-hystrix-fallback-stream。
5. 启动项目 microservice-hystrix-turbine。
6. 启动项目 microservice-hystrix-dashboard。
7. 访问*http://localhost:8010/user/1*，让 microservice-consumer-movie-ribbon-hystrix 微服务产生监控数据。
8. 访问*http://localhost:8020/user/1*，让 microservice-consumer-movie-feign-hystrix-fallback-stream 微服务产生监控数据。
9. 打开 Hystrix Dashboard 首页*http://localhost:8030/hystrix.stream*，在 URL 一栏填入*http://localhost:8031/turbine.stream*，随意指定一个 Title 并点击 Monitor Stream 按钮后，结果类似于图 7-7。

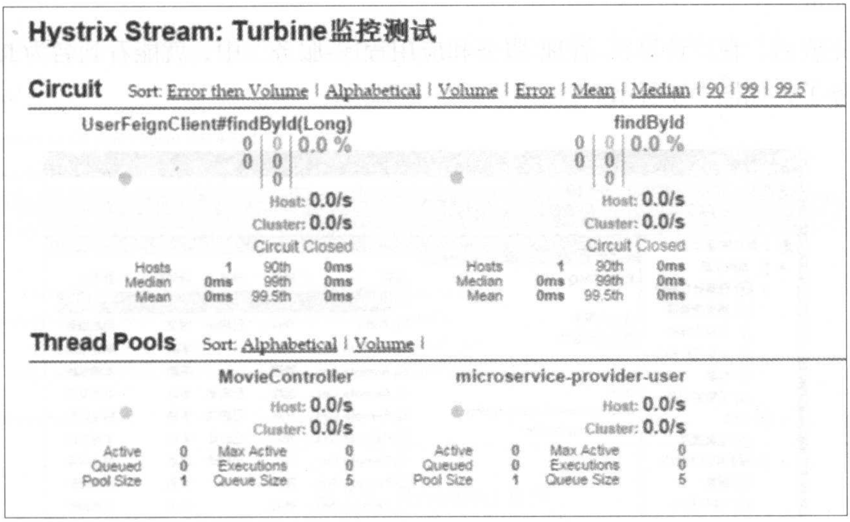


图 7-7 Turbine 监控多个微服务

7.5.3 使用消息中间件收集数据

一些场景下，前文的方式无法正常工作（例如微服务与 Turbine 网络不通），此时，可借助消息中间件实现数据收集。各个微服务将 Hystrix Command 的监控数据发送至消息中间件，Turbine 消费消息中间件中的数据。

下面笔者以 RabbitMQ 为例进行演示。

7.5.3.1 安装 RabbitMQ

先来安装 RabbitMQ，以 Windows 操作系统为例。

一、安装 RabbitMQ

1. 安装 Erlang/OTP 19.2

   RabbitMQ 依赖 ERlang，先来安装 ERlang。

   通过官方下载页面：*http://www.erlang.org/downloads*，获取 exe 安装包，双击打开，按照提示即可完成安装。

2. 安装 RabbitMQ Server 3.6.6

   通过官方下载页面*http://www.rabbitmq.com/install-windows.html*，获取 exe 安装包，双击打开，按照提示即可完成安装。

3. 安装完成

   安装完成后，在“计算机-管理-服务和应用程序-服务”中，就能看到名为 RabbitMQ 的服务了，如图 7-8 所示。

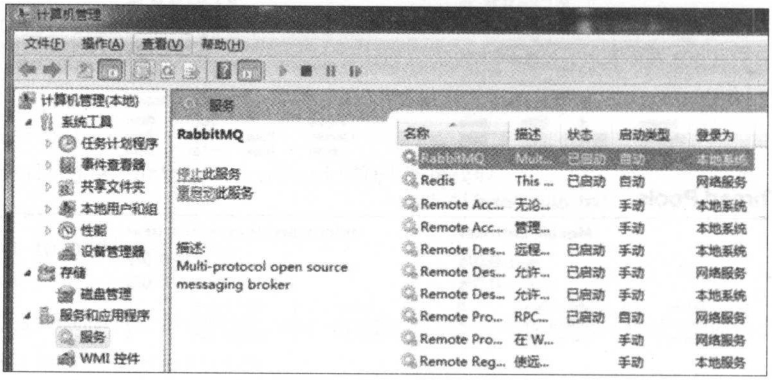


图 7-8 RabbitMQ 注册为服务

二、安装 RabbitMQ 管理插件

为了更加方便地管理 RabbitMQ，接着安装 RabbitMQ 的管理插件。

1. 将目录切换到 RabbitMQ 中的 sbin 目录，例如：

```
cd D:\Program Files\RabbitMQ Server\rabbitmq_server-3.6.6\sbin>
```

   当然，也可点击开始菜单中的“RabbitMQ Command Prompt (sbin dir)”菜单，直接切换到 sbin 目录。

2. 执行以下命令，安装管理插件。

```
rabbitmq-plugins enable rabbitmq_management
```

3. 访问*http://localhost:15672/*，输入默认账号 guest，密码 guest，即可看到如图 7-9 的界面。

图 7-9 RabbitMQ 首页

这样就可使用图形化的界面管理 RabbitMQ 了。

7.5.3.2 改造微服务

要想使用消息中间件收集监控数据，需要改造前文编写的微服务，以 microservice-consumer-movie-ribbon-hystrix 为例。

1. 复制项目`microservice-consumer-movie-ribbon-hystrix`，将 ArtifactId 修改为 `microservice-consumer-movie-ribbon-hystrix-turbine-mq`。
2. 添加以下依赖。

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-netflix-hystrix-stream</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

3. 在配置文件 application.yml 中添加如下内容，连接 RabbitMQ。

```
spring:
  rabbitmq:
    host: localhost
    port: 5672
    username: guest
    password: guest
```

这样，微服务就改造完成了。

7.5.3.3 改造 Turbine

改造完微服务后，接下来改造 Turbine。

1. 复制项目microservice-hystrix-turbine，将 ArtifactId 修改为 microservice-hystrix-turbine-mq。
2. 在 pom.xml 中，添加如下内容。

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-turbine-stream</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

同时，删除：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-turbine</artifactId>
</dependency>
```

3. 修改启动类，将注解`@EnableTurbine`修改为`@EnableTurbineStream`。
4. 修改配置文件 application.yml，添加如下内容。

```
spring:
  rabbitmq:
    host: localhost
    port: 5672
    username: guest
    password: guest
```

同时，删除：

```
turbine:
  appConfig: microservice-consumer-movie,microservice-consumer-movie-feign-
      hystrix-fallback-stream
  clusterNameExpression: "'default'"
```

这样，Turbine 就改造完成了。

测试

1. 启动项目 microservice-discovery-eureka。
2. 启动项目 microservice-provider-user。
3. 启动项目 microservice-consumer-movie-ribbon-hystrix-turbine-mq。
4. 启动项目 microservice-hystrix-turbine-mq。
5. 访问*http://localhost:8010/user/1*，可正常获得结果。
6. 访问*http://localhost:8031/*，会发现 Turbine 能够持续不断地显示监控数据。

在 Spring Cloud Camden SR4 中，依赖 spring-cloud-starter-turbine 不能与 spring-cloud-starter-turbine-stream 共存，否则启动时会报异常。不仅如此，这两个依赖使用的 Turbine 版本也不相同，spring-cloud-starter-turbine 使用的 Turbine 版本是 1.0.0，而 spring-cloud-starter-turbine-stream 使用的 Turbine 版本是 2.0.0-DP.2。笔者已在 GitHub 上提出相关 Issue，详见*https://github.com/spring-cloud/spring-cloud-netflix/issues/1629*。

8

使用 Zuul 构建微服务网关

8.1 为什么要使用微服务网关

经过前文的讲解，微服务架构已经初具雏形，但还有一些问题——不同的微服务一般会有不同的网络地址，而外部客户端（例如手机 APP）可能需要调用多个服务的接口才能完成一个业务需求。例如一个电影购票的手机 APP，可能会调用多个微服务的接口，才能完成一次购票的业务流程，如图 8-1 所示。

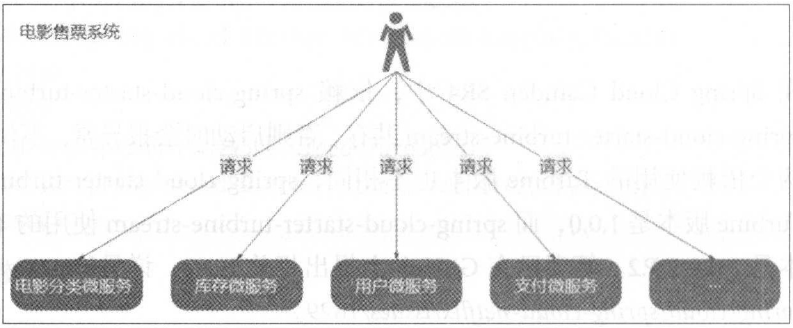


图 8-1　用户请求多个微服务

如果让客户端直接与各个微服务通信，会有以下的问题：

- 客户端会多次请求不同的微服务，增加了客户端的复杂性。
- 存在跨域请求，在一定场景下处理相对复杂。
- 认证复杂，每个服务都需要独立认证。
- 难以重构，随着项目的迭代，可能需要重新划分微服务。例如，可能将多个服务合并成一个或者将一个服务拆分成多个。如果客户端直接与微服务通信，那么重构将会很难实施。
- 某些微服务可能使用了防火墙/浏览器不友好的协议，直接访问会有一定的困难。

以上问题可借助微服务网关解决。微服务网关是介于客户端和服务器端之间的中间层，所有的外部请求都会先经过微服务网关。使用微服务网关后，架构可演变成图 8-2。

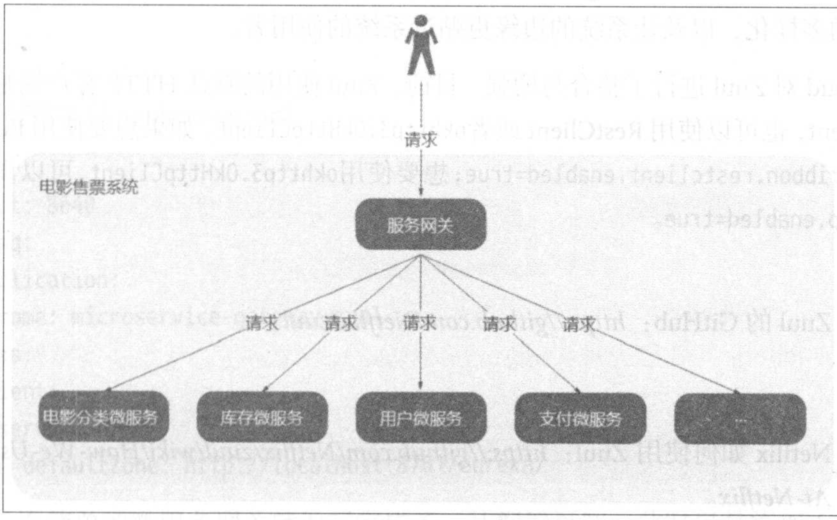


图 8-2 使用微服务网关

如图，微服务网关封装了应用程序的内部结构，客户端只须跟网关交互，而无须直接调用特定微服务的接口。这样，开发就可以得到简化。不仅如此，使用微服务网关还有以下优点：

- 易于监控。可在微服务网关收集监控数据并将其推送到外部系统进行分析。
- 易于认证。可在微服务网关上进行认证，然后再将请求转发到后端的微服务，而无须在每个微服务中进行认证。
- 减少了客户端与各个微服务之间的交互次数。

8.2 Zuul 简介

Zuul 是 Netflix 开源的微服务网关，它可以和 Eureka、Ribbon、Hystrix 等组件配合使用。Zuul 的核心是一系列的过滤器，这些过滤器可以完成以下功能。

- 身份认证与安全：识别每个资源的验证要求，并拒绝那些与要求不符的请求。
- 审查与监控：在边缘位置追踪有意义的数据和统计结果，从而带来精确的生产视图。
- 动态路由：动态地将请求路由到不同的后端集群。
- 压力测试：逐渐增加指向集群的流量，以了解性能。
- 负载分配：为每一种负载类型分配对应容量，并弃用超出限定值的请求。
- 静态响应处理：在边缘位置直接建立部分响应，从而避免其转发到内部集群。
- 多区域弹性：跨越 AWS Region 进行请求路由，旨在实现 ELB（Elastic Load Balancing）使用的多样化，以及让系统的边缘更贴近系统的使用者。

Spring Cloud 对 Zuul 进行了整合与增强。目前，Zuul 使用的默认 HTTP 客户端是 Apache HTTP Client，也可以使用 RestClient 或者`okhttp3.OkHttpClient`。如果想要使用 RestClient，可以设置`ribbon.restclient.enabled=true`；想要使用`okhttp3.OkHttpClient`，可以设置 `ribbon.okhttp.enabled=true`。

Zuul 的 GitHub：*https://github.com/Netflix/zuul*。

Netflix 如何使用 Zuul：*https://github.com/Netflix/zuul/wiki/How-We-Use-Zuul-At-Netflix*。

8.3 编写 Zuul 微服务网关

本节将编写一个简单的微服务网关。在本例中会将 Zuul 注册到 Eureka Server 上。

1. 创建一个 Maven 工程，ArtifactId 是`microservice-gateway-zuul`，并为项目添加以下依赖。

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-zuul</artifactId>
```

```
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
```

2. 在启动类上添加注解`@EnableZuulProxy`，声明一个 Zuul 代理。该代理使用 Ribbon 来定位注册在 Eureka Server 中的微服务；同时，该代理还整合了 Hystrix，从而实现了容错，所有经过 Zuul 的请求都会在 Hystrix 命令中执行。

```
@SpringBootApplication
@EnableZuulProxy
public class ZuulApplication {
  public static void main(String[] args) {
    SpringApplication.run(ZuulApplication.class, args);
  }
}
```

3. 编写配置文件 application.yml。

```
server:
  port: 8040
spring:
  application:
    name: microservice-gateway-zuul
eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/
```

这样，一个简单的微服务网关就编写完成了。从配置可知，此时仅是添加了 Zuul 的依赖，并将 Zuul 注册到 Eureka Server 上。

测试：路由规则

1. 启动项目 microservice-discovery-eureka。
2. 启动项目 microservice-provider-user。
3. 启动项目 microservice-consumer-movie-ribbon。
4. 启动项目 microservice-gateway-zuul。
5. 访问*http://localhost:8040/microservice-consumer-movie/user/1*，请求会被转发到*http:*

//localhost:8010/user/1（电影微服务）。

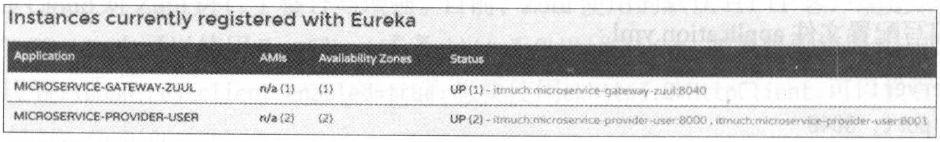
6. 访问*http://localhost:8040/microservice-provider-user/1*，请求会被转发到*http://localhost:8000/1*（用户微服务）。

说明默认情况下，Zuul 会代理所有注册到 Eureka Server 的微服务，并且 Zuul 的路由规则如下：

`http://ZUUL_HOST:ZUUL_PORT/微服务在Eureka上的serviceId/**` 会被转发到 serviceId 对应的微服务。

测试：负载均衡

1. 启动项目 microservice-discovery-eureka。
2. 启动多个 microservice-provider-user 实例。
3. 启动项目 microservice-gateway-zuul。此时，Eureka Server 首页如图 8-3 所示。



Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
MICROSERVICE-GATEWAY-ZUUL	n/a (1)	(1)	UP (1) - itmuch:microservice-gateway-zuul:8040
MICROSERVICE-PROVIDER-USER	n/a (2)	(2)	UP (2) - itmuch:microservice-provider-user:8000 , itmuch:microservice-provider-user:8001

图 8-3 Eureka Server 上的微服务列表

4. 多次访问*http://localhost:8040/microservice-provider-user/1*，会发现两个用户微服务节点都会打印类似以下的日志。

```
Hibernate: select user0_.id as id1_0_0_, user0_.age as age2_0_0_, user0_.
balance as balance3_0_0_, user0_.name as name4_0_0_, user0_.username as
username5_0_0_ from user user0_ where user0_.id=?
```

说明 Zuul 可以使用 Ribbon 达到负载均衡的效果。

测试：Hystrix 容错与监控

1. 启动项目 microservice-discovery-eureka。
2. 启动项目 microservice-provider-user。
3. 启动项目 microservice-consumer-movie-ribbon。
4. 启动项目 microservice-gateway-zuul。

5. 启动项目 microservice-hystrix-dashboard。
6. 访问*http://localhost:8040/microservice-consumer-movie/user/1*，能获得预期结果。
7. Hystrix Dashboard 中输入 *http://localhost:8040/hystrix.stream*，随意指定一个 Title 并点击 Monitor Stream 按钮后，结果如图 8-4 所示。

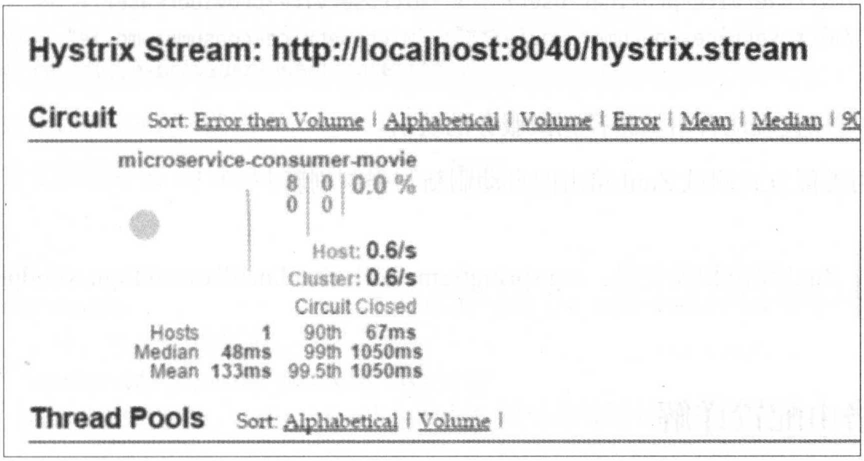


图 8-4 Hystrix Dashboard 监控页面

说明 Zuul 已经整合了 Hystrix。

8.4 Zuul的路由端点

当`@EnableZuulProxy`与 Spring Boot Actuator 配合使用时，Zuul 会暴露一个路由管理端点/`routes`。借助这个端点，可以方便、直观地查看以及管理 Zuul 的路由。

/routes 端点的使用非常简单，使用 GET 方法访问该端点，即可返回 Zuul 当前映射的路由列表；使用 POST 方法访问该端点就会强制刷新 Zuul 当前映射的路由列表（尽管路由会自动刷新，Spring Cloud 依然提供了强制立即刷新的方式）。

由于`spring-cloud-starter-zuul`已经包含了`spring-boot-starter-actuator`，因此之前编写的`microservice-gateway-zuul`已具备路由管理的能力。

下面来做一些测试。

1. 启动 microservice-discovery-eureka。
2. 启动 microservice-provider-user。

3. 启动 microservice-consumer-movie。
4. 启动 microservice-gateway-zuul。
5. 使用浏览器访问*http://localhost:8040/routes*，可获得如下的结果。

```
{
  "/microservice-provider-user/**": "microservice-provider-user",
  "/microservice-consumer-movie/**": "microservice-consumer-movie"
}
```

从中可以直观地看出从路径到微服务的映射。

也可使用类似方式测试 Zuul 路由的自动刷新与强制刷新。

Zuul 路由相关代码：org.springframework.cloud.netflix.zuul.RoutesEndpoint。

8.5 路由配置详解

前文已经编写了一个简单的 Zuul 网关，并让该网关代理了所有注册到 Eureka Server 的微服务。但在现实中可能只想让 Zuul 代理部分微服务，又或者需要对 URL 进行更加精确的控制。

Zuul 的路由配置非常灵活、简单，本节通过几个示例，详细讲解 Zuul 的路由配置。

1. 自定义指定微服务的访问路径。

   配置`zuul.routes.指定微服务的serviceId = 指定路径`即可。例如：

```
zuul:
  routes:
    microservice-provider-user: /user/**
```

   这样设置，microservice-provider-user 微服务就会被映射到/user/** 路径。

2. 忽略指定微服务。

   忽略服务非常简单，可以使用`zuul.ignored-services`配置需要忽略的服务，多个用逗号分隔。例如：

```
zuul:
  ignored-services: microservice-provider-user,microservice-consumer-movie
```

   这样就可让 Zuul 忽略 microservice-provider-user 和 microservice-consumer-movie 微服务，只代理其他微服务。

3. 忽略所有微服务，只路由指定微服务。

很多场景下，可能只想要让 Zuul 代理指定的微服务，此时可以将`zuul.ignored-services`设为'*'。

```
zuul:
  ignored-services: '*'   # 使用'*'可忽略所有微服务
  routes:
    microservice-provider-user: /user/**
```

这样就可以让 Zuul 只路由 microservice-provider-user 微服务。

4. 同时指定微服务的 serviceId 和对应路径。例如：

```
zuul:
  routes:
    user-route:                   # 该配置方式中，user-route只是给路由一个名
      称，可以任意起名。
      service-id: provider-microservice-user
      path: /user/**              # service-id对应的路径
```

本例配置的效果同示例 1。

5. 同时指定 path 和 URL，例如：

```
zuul:
  routes:
    user-route:                   # 该配置方式中，user-route只是给路由一个名
      称，可以任意起名。
      url: http://localhost:8000/ # 指定的url
      path: /user/**              # url对应的路径。
```

这样就可以将`/user/**`映射到`http://localhost:8000/**`。

需要注意的是，使用这种方式配置的路由不会作为 HystrixCommand 执行，同时也不能使用 Ribbon 来负载均衡多个 URL。例 6 可解决该问题。

6. 同时指定 path 和 URL，并且不破坏 Zuul 的 Hystrix、Ribbon 特性。

```
zuul:
  routes:
    user-route:
      path: /user/**
      service-id: microservice-provider-user
ribbon:
  eureka:
```

```
    enabled: false    # 为Ribbon禁用Eureka
microservice-provider-user:
  ribbon:
    listOfServers: localhost:8000,localhost:8001
```

这样就可以既指定 path 与 URL，又不破坏 Zuul 的 Hystrix 与 Ribbon 特性了。

7. 使用正则表达式指定 Zuul 的路由匹配规则

借助 PatternServiceRouteMapper，实现从微服务到映射路由的正则配置。例如：

```
@Bean
public PatternServiceRouteMapper serviceRouteMapper() {
  // 调用构造函数PatternServiceRouteMapper(String servicePattern, String
      routePattern)
  // servicePattern指定微服务的正则
  // routePattern指定路由正则
  return new PatternServiceRouteMapper("(?<name>^.+)-(?<version>v.+$)", "${
      version}/${name}");
}
```

通过这段代码即可实现将诸如 microservice-provider-user-v1 这个微服务，映射到/v1/microservice-provider-user/** 这个路径。

8. 路由前缀

示例 1：

```
zuul:
  prefix: /api
  strip-prefix: false
  routes:
    microservice-provider-user: /user/**
```

这样，访问 Zuul 的/api/microservice-provider-user/1 路径，请求将会被转发到 microservice-provider-user 的/api/1

示例 2：

```
zuul:
  routes:
    microservice-provider-user:
      path: /user/**
      strip-prefix: false
```

这样访问 Zuul 的/user/1 路径，请求将会被转发到 microservice-provider-user 的/user/1。

- 可参考该 Issue 辅助理解：*https://github.com/spring-cloud/spring-cloud-netflix/issues/1365*。
- 该特性可能有 Bug，相关 Issue：*https://github.com/spring-cloud/spring-cloud-netflix/issues/1514*。

9. 忽略某些路径

上文讲解了如何忽略微服务，但有时还需要更细粒度的路由控制。例如，想让 Zuul 代理某个微服务，同时又想保护该微服务的某些敏感路径。此时，可使用 ignored-Patterns，指定忽略的正则。例如：

```
zuul:
  ignoredPatterns: /**/admin/**   # 忽略所有包含/admin/的路径
  routes:
    microservice-provider-user: /user/**
```

这样就可将 microservice-provider-user 微服务映射到/user/** 路径，但会忽略该微服务中所有包含/admin/的路径。

读者如无法掌握 Zuul 路由的规律，可将 com.netflix 包的日志级别设为 DEBUG。这样，Zuul 就会打印转发的具体细节，从而有助于更好地理解 Zuul 的路由配置，例如：

```
logging:
  level:
    com.netflix: DEBUG
```

8.6 Zuul 的安全与 Header

本节来讨论 Zuul 的安全与 Header。

8.6.1 敏感 Header 的设置

一般来说，可在同一个系统中的服务之间共享 Header。不过应尽量防止让一些敏感的 Header 外泄。因此，在很多场景下，需要通过为路由指定一系列敏感 Header 列表。例如：

```
zuul:
  routes:
```

```
  microservice-provider-user:
    path: /users/**
    sensitive-headers: Cookie,Set-Cookie,Authorization
    url: https://downstream
```

这样就可为 microservice-provider-user 微服务指定敏感 Header 了。

也可用`zuul.sensitive-headers`全局指定敏感 Header，例如：

```
zuul:
  sensitive-headers: Cookie,Set-Cookie,Authorization # 默认是Cookie,Set-Cookie,
    Authorization
```

需要注意的是，如果使用`zuul.routes.*.sensitive-headers`的配置方式，会覆盖掉全局的配置。

8.6.2 忽略 Header

可使用`zuul.ignoredHeaders`属性丢弃一些 Header，例如：

```
zuul:
  ignored-headers: Header1, Header2
```

这样设置后，Header1 和 Header2 将不会传播到其他的微服务中。

默认情况下，zuul.ignored-headers 是空值，但如果 Spring Security 在项目的 classpath 中，那么 zuul.ignored-headers 的默认值就是`Pragma,Cache-Control,X-Frame-Options,X-Content-Type-Options,X-XSS-Protection,Expires`。所以，当 Spring Security 在项目的 classpath 中，同时又需要使用下游微服务的 Spring Security 的 Header 时，可以将 `zuul.ignoreSecurityHeaders` 设置为 false。

- 笔者在 Github 上提的 Issue：*https://github.com/spring-cloud/spring-cloud-netflix/issues/1487*，希望可以帮助大家理解。
- sensitive-headers 与 ignored-headers 的单元测试：*https://github.com/spring-cloud/spring-cloud-netflix/blob/master/spring-cloud-netflix-core/src/test/java/org/springframework/cloud/netflix/zuul/filters/pre/PreDecorationFilterTests.java*。
- 事实上，sensitive-headers 会被添加到 ignored-headers 中，详见代码 org.springframework.cloud.netflix.zuul.filters.ProxyRequestHelper.addIgnoredHeaders(String...) 和 org.springframework.cloud.netflix.zuul.filters.pre.PreDecorationFilter.run()。

8.7 使用 Zuul 上传文件

本节来讨论 Zuul 的文件上传。

对于小文件（1M 以内）上传，无须任何处理，即可正常上传。对于大文件（10M 以上）上传，需要为上传路径添加`/zuul`前缀。也可使用`zuul.servlet-path`自定义前缀。

也就是说，假设`zuul.routes.microservice-file-upload = /microservice-file-upload/**`，如果`http://{HOST}:{PORT}/upload`是微服务 microservice-file-upload 的上传路径，则可使用 Zuul 的`/zuul/microservice-file-upload/upload`路径上传大文件。

如果 Zuul 使用了 Ribbon 做负载均衡，那么对于超大的文件（例如 500M），需要提升超时设置，例如：

```
hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds: 60000
ribbon:
  ConnectTimeout: 3000
  ReadTimeout: 60000
```

下面编写一个文件上传的微服务，详细讲解 Zuul 的文件上传。

编写文件上传微服务

1. 创建一个 Maven 工程，ArtifactId 是`microservice-file-upload`，并为项目添加以下依赖。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

2. 在启动类上添加`@SpringBootApplication`、`@EnableEurekaClient`注解。
3. 编写 Controller。

```
@Controller
public class FileUploadController {
  /**
   * 上传文件
   * 测试方法：
   * 有界面的测试：http://localhost:8050/index.html
   * 使用命令：curl -F "file=@文件全名" localhost:8050/upload
   * ps.该示例比较简单，没有做IO异常、文件大小、文件非空等处理
   * @param file 待上传的文件
   * @return 文件在服务器上的绝对路径
   * @throws IOException IO异常
   */
  @RequestMapping(value = "/upload", method = RequestMethod.POST)
  public @ResponseBody String handleFileUpload(@RequestParam(value = "file",
      required = true) MultipartFile file) throws IOException {
    byte[] bytes = file.getBytes();
    File fileToSave = new File(file.getOriginalFilename());
    FileCopyUtils.copy(bytes, fileToSave);
    return fileToSave.getAbsolutePath();
  }
}
```

4. 配置文件 application.yml，在其中添加如下内容。

```
server:
  port: 8050
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
  instance:
    prefer-ip-address: true
spring:
  application:
    name: microservice-file-upload
  http:
    multipart:
      max-file-size: 2000Mb      # Max file size，默认1M
      max-request-size: 2500Mb   # Max request size，默认10M
```

由配置可知，已经将该服务注册到 Eureka Server 上，并配置了文件上传大小的限制。

这样一个文件上传的微服务就编写完成了。可使用以下命令测试文件上传。

```
curl -F "file=@文件全名" localhost:8050/upload
```

测试

笔者使用的测试工具是 cURL，大家也可使用其他工具进行测试。

1. 准备 1 个小文件（1M 以下），记为 small.file；1 个超大文件（1G 以上，2G 以上），记为 large.file。
2. 启动 microservice-discovery-eureka。
3. 启动 microservice-file-upload。
4. 启动 microservice-gateway-zuul。
5. 测试直接上传到 microservice-file-upload 上：使用命令`curl -F "file=@large.file" localhost:8050/upload`，上传大文件，发现可以正常上传。同理，小文件也可以上传。
6. 测试通过 Zuul 上传小文件。

   ```
   curl -v -H "Transfer-Encoding: chunked" -F "file=@small.file" localhost:
   8040/microservice-file-upload/upload
   ```

   发现可以正常上传。
7. 测试通过 Zuul 上传大文件，不添加/zuul 前缀。

   ```
   curl -v -H "Transfer-Encoding: chunked" -F "file=@large.file" localhost:
   8040/microservice-file-upload/upload
   ```

   发现 Zuul 会报类似以下异常：

   ```
   org.apache.tomcat.util.http.fileupload.FileUploadBase$FileSizeLimitExceed
   edException: The field file exceeds its maximum permitted size of 1048576
   bytes.
   ```
8. 测试通过 Zuul 上传大文件，添加/zuul 前缀。

   ```
   curl -v -H "Transfer-Encoding: chunked" -F "file=@large.file" localhost:
   8040/zuul/microservice-file-upload/upload
   ```

   此时，Zuul 会报以下异常：

   ```
   Caused by: com.netflix.hystrix.exception.HystrixRuntimeException: micro
   service-file-upload timed-out and no fallback available.
   ```

可以看到，此时已经不是文件大小限制的异常了，而只是 Hystrix 的超时异常。

9. 因此，不妨在 application.yml 中添加如下内容。

```
# 上传大文件得将超时时间设置长一些，否则会报超时异常。以下几行超时设
  置来自http://cloud.spring.io/spring-cloud-static/Camden.SR4/#
  _uploading_files_through_zuul
hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds:
  60000
ribbon:
  ConnectTimeout: 3000
  ReadTimeout: 60000
```

详见本书配套代码中的 microservice-gateway-zuul-file-upload 项目。

10. 关闭microservice-gateway-zuul，启动microservice-gateway-zuul-file-upload再次使用。

```
curl -v -H "Transfer-Encoding: chunked" -F "file=@large.file" localhost:
8040/zuul/microservice-file-upload/upload
```

此时已可正常上传文件。

8.8 Zuul 的过滤器

过滤器是 Zuul 的核心组件，本节来详细讨论 Zuul 的过滤器。

8.8.1 过滤器类型与请求生命周期

Zuul 大部分功能都是通过过滤器来实现的。Zuul 中定义了 4 种标准过滤器类型，这些过滤器类型对应于请求的典型生命周期。

- PRE：这种过滤器在请求被路由之前调用。可利用这种过滤器实现身份验证、在集群中选择请求的微服务、记录调试信息等。
- ROUTING：这种过滤器将请求路由到微服务。这种过滤器用于构建发送给微服务的请求，并使用 Apache HttpClient 或 Netfilx Ribbon 请求微服务。
- POST：这种过滤器在路由到微服务以后执行。这种过滤器可用来为响应添加标准的 HTTP Header、收集统计信息和指标、将响应从微服务发送给客户端等。
- ERROR：在其他阶段发生错误时执行该过滤器。

除了默认的过滤器类型，Zuul 还允许创建自定义的过滤器类型。例如，可以定制一种 STATIC 类型的过滤器，直接在 Zuul 中生成响应，而不将请求转发到后端的微服务。

Zuul 请求的生命周期如图 8-5 所示，该图详细描述了各种类型的过滤器的执行顺序。

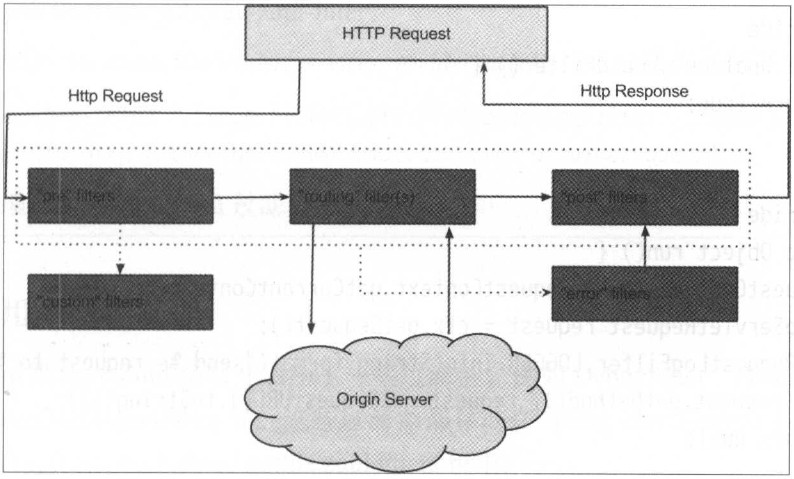


图 8-5 Zuul 请求的生命周期

8.8.2 编写 Zuul 过滤器

理解过滤器类型和请求生命周期后，来编写一个 Zuul 过滤器。编写 Zuul 的过滤器非常简单，只须继承抽象类 ZuulFilter，然后实现几个抽象方法就可以了。

那么现在来编写一个简单的 Zuul 过滤器，让该过滤器打印请求日志。

1. 复制项目`microservice-gateway-zuul`，将 ArtifactId 修改为`microservice-gateway-zuul-filter`。
2. 编写自定义 Zuul 过滤器。

```
public class PreRequestLogFilter extends ZuulFilter {
  private static final Logger LOGGER = LoggerFactory.getLogger(
      PreRequestLogFilter.class);

  @Override
  public String filterType() {
    return "pre";
  }
```

```
  @Override
  public int filterOrder() {
    return 1;
  }

  @Override
  public boolean shouldFilter() {
    return true;
  }

  @Override
  public Object run() {
    RequestContext ctx = RequestContext.getCurrentContext();
    HttpServletRequest request = ctx.getRequest();
    PreRequestLogFilter.LOGGER.info(String.format("send %s request to %s",
        request.getMethod(), request.getRequestURL().toString()));
    return null;
  }
}
```

由代码可知，自定义的 Zuul Filter 需实现以下几个方法。

- filterType：返回过滤器的类型。有 pre、route、post、error 等几种取值，分别对应上文的几种过滤器。详细可以参考com.netflix.zuul.ZuulFilter.filterType()中的注释。
- filterOrder：返回一个 int 值来指定过滤器的执行顺序，不同的过滤器允许返回相同的数字。
- shouldFilter：返回一个 boolean 值来判断该过滤器是否要执行，true 表示执行，false 表示不执行。
- run：过滤器的具体逻辑。本例中让它打印了请求的 HTTP 方法以及请求的地址。

3. 修改启动类，为启动类添加以下内容：

```
@Bean
public PreRequestLogFilter preRequestLogFilter() {
  return new PreRequestLogFilter();
}
```

测试

1. 启动 microservice-discovery-eureka。
2. 启动 microservice-provider-user。
3. 启动 microservice-gateway-zuul-filter。
4. 访问*http://localhost:8040/microservice-provider-user/1*，可获得类似如下的日志。

```
[nio-8040-exec-6] c.i.c.s.filters.pre.PreRequestLogFilter  : send GET
request to http://localhost:8040//microservice-provider-user/1
```

说明编写的自定义 Zuul 过滤器被执行了。

8.8.3 禁用 Zuul 过滤器

Spring Cloud 默认为 Zuul 编写并启用了一些过滤器，例如 DebugFilter、FormBodyWrapperFilter、PreDecorationFilter 等。这些过滤器都存放在 spring-cloud-netflix-core 这个 Jar 包的 org.springframework.cloud.netflix.zuul.filters 包中。

一些场景下，想要禁用掉部分过滤器，该怎么办呢？

答案非常简单，只须设置`zuul.<SimpleClassName>.<filterType>.disable=true`，即可禁用 SimpleClassName 所对应的过滤器。以过滤器 org.springframework.cloud.netflix.zuul.filters.post.SendResponseFilter 为例，只须设置`zuul.SendResponseFilter.post.disable=true`，即可禁用该过滤器。

同理，如果想要禁用 8.8.2 节编写的过滤器，只需设置 `zuul.PreRequestLogFilter.pre.disable=true` 即可。

相关代码：com.netflix.zuul.ZuulFilter.disablePropertyName()、com.netflix.zuul.ZuulFilter.isFilterDisabled()、com.netflix.zuul.ZuulFilter.runFilter()。

8.9 Zuul 的容错与回退

在 Spring Cloud 中，Zuul 默认已经整合了 Hystrix。首先来做一个简单的实验：

1. 启动项目 microservice-discovery-eureka。
2. 启动项目 microservice-provider-user。

3. 启动项目 microservice-gateway-zuul。
4. 启动项目 microservice-hystrix-dashboard。
5. 访问*http://localhost:8040/microservice-provider-user/1*，可正常获得结果。
6. 访问*http://localhost:8040/hystrix.stream*，可获得 Hystrix 的监控数据。
7. 访问*http://localhost:8030/hystrix*，并在监控地址一栏填入*http://localhost:8040/hystrix.stream*，即可看到类似图 8-6 的图表。

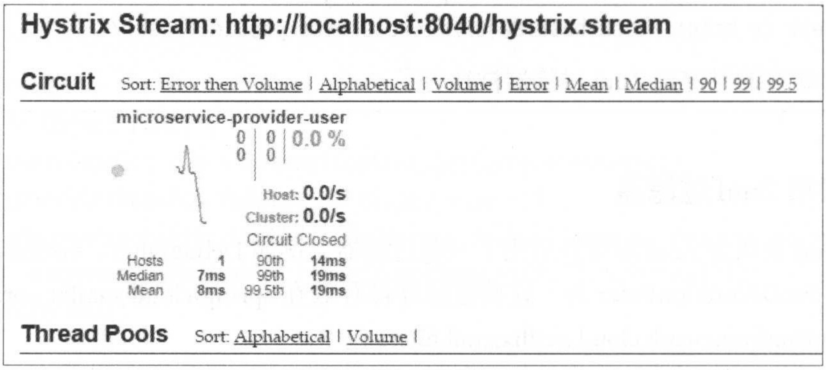


图 8-6 Hystrix Dashboard 监控页面

由图可知，Zuul 的 Hystrix 监控的粒度是微服务，而不是某个 API；同时也说明，所有经过 Zuul 的请求，都会被 Hystrix 保护起来。

8. 关闭项目 microservice-provider-user，再次访问*http://localhost:8040/microservice-provider-user/1*，将会看到页面输出类似于如下的异常：

```
Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as
a fallback.

Thu Nov 24 15:28:20 CST 2016
There was an unexpected error (type=Internal Server Error, status=500).
TIMEOUT
```

下面来谈谈如何为 Zuul 实现回退。

为 Zuul 添加回退

想要为 Zuul 添加回退，需要实现 ZuulFallbackProvider 接口。在实现类中，指定为哪个微服务提供回退，并提供一个 ClientHttpResponse 作为回退响应。

下面来编写代码。

1. 复制项目`microservice-gateway-zuul`，将 ArtifactId 修改为`microservice-gateway-zuul-fallback`。
2. 编写 Zuul 的回退类：

```
@Component
public class UserFallbackProvider implements ZuulFallbackProvider {
  @Override
  public String getRoute() {
    // 表明是为哪个微服务提供回退
    return "microservice-provider-user";
  }

  @Override
  public ClientHttpResponse fallbackResponse() {
    return new ClientHttpResponse() {
      @Override
      public HttpStatus getStatusCode() throws IOException {
        // fallback时的状态码
        return HttpStatus.OK;
      }

      @Override
      public int getRawStatusCode() throws IOException {
        // 数字类型的状态码，本例返回的其实就是200，详见HttpStatus
        return this.getStatusCode().value();
      }

      @Override
      public String getStatusText() throws IOException {
        // 状态文本，本例返回的其实就是OK，详见HttpStatus
        return this.getStatusCode().getReasonPhrase();
      }

      @Override
      public void close() {
      }

      @Override
```

```
      public InputStream getBody() throws IOException {
        // 响应体
        return new ByteArrayInputStream("用户微服务不可用，请稍后再试。".
            getBytes());
      }

      @Override
      public HttpHeaders getHeaders() {
        // headers设定
        HttpHeaders headers = new HttpHeaders();
        MediaType mt = new MediaType("application","json",
                Charset.forName("UTF-8"));
        headers.setContentType(mt);
        return headers;
      }
    };
  }
}
```

添加回退之后，重复之前的实验，当 microservice-provider-user 微服务无法正常响应时，将会返回以下内容。

```
用户微服务不可用，请稍后再试。
```

8.10 Zuul的高可用

Zuul 的高可用非常关键，因为外部请求到后端微服务的流量都会经过 Zuul。故而在生产环境中一般都需要部署高可用的 Zuul 以避免单点故障。

笔者分两种场景讨论 Zuul 的高可用。

8.10.1 Zuul 客户端也注册到了 Eureka Server 上

这种情况下，Zuul 的高可用非常简单，只须将多个 Zuul 节点注册到 Eureka Server 上，就可实现 Zuul 的高可用。此时，Zuul 的高可用与其他微服务的高可用没什么区别。

如图 8-7，当 Zuul 客户端也注册到 Eureka Server 上时，只须部署多个 Zuul 节点即可实现其高可用。Zuul 客户端会自动从 Eureka Server 中查询 Zuul Server 的列表，并使用 Ribbon 负载均衡地请求 Zuul 集群。

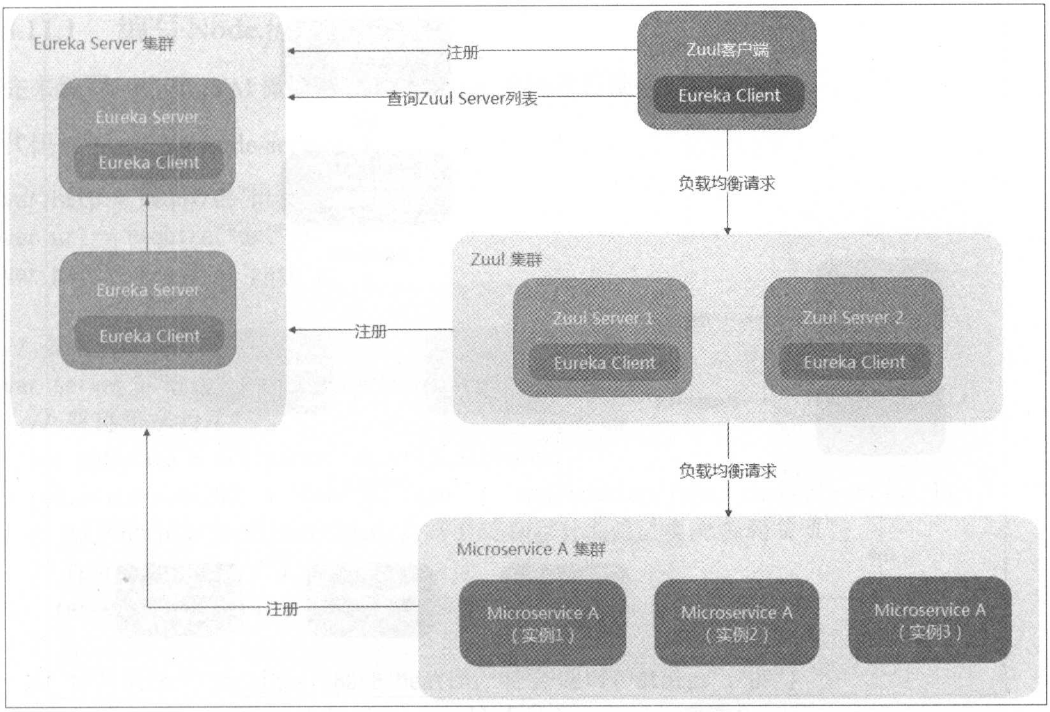


图 8-7　Zuul 高可用架构图

8.10.2　Zuul 客户端未注册到 Eureka Server 上

现实中，这种场景往往更常见，例如，Zuul 客户端是一个手机 APP——不可能让所有的手机终端都注册到 Eureka Server 上。这种情况下，可借助一个额外的负载均衡器来实现 Zuul 的高可用，例如 Nginx、HAProxy、F5 等。

如图 8-8，Zuul 客户端将请求发送到负载均衡器，负载均衡器将请求转发到其代理的其中一个 Zuul 节点。这样，就可以实现 Zuul 的高可用。

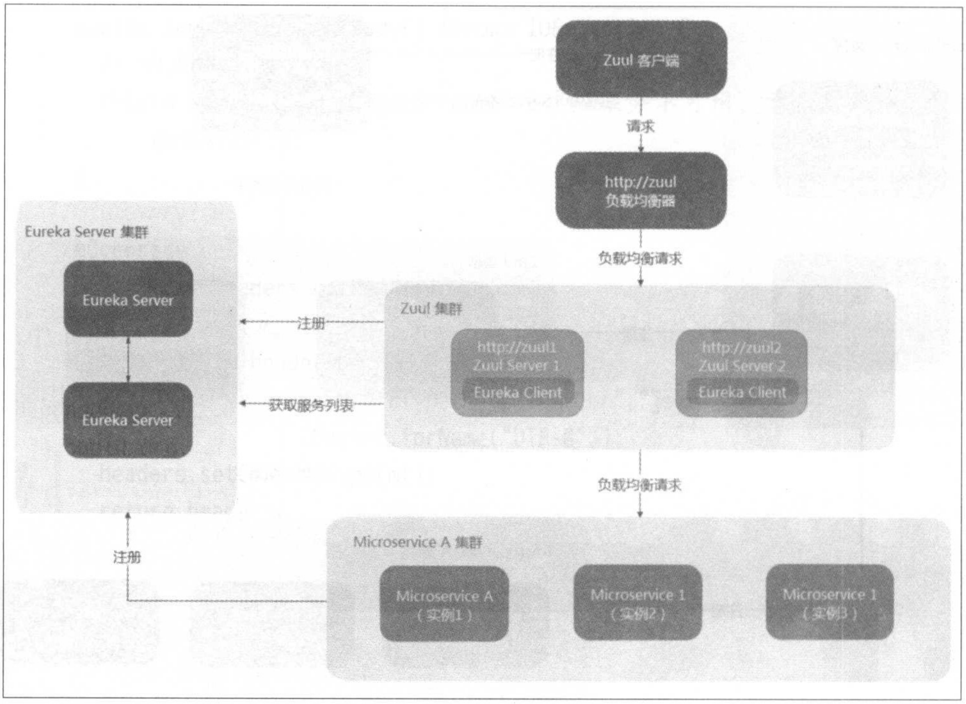


图 8-8 Zuul 高可用架构图

8.11 使用 Sidecar 整合非 JVM 微服务

在 4.9 节，笔者曾经提到，非 JVM 微服务可操作 Eureka 的 REST 端点，从而实现注册与发现。事实上，也可使用 Sidecar（挎斗）更加方便地整合非 JVM 微服务。

Spring Cloud Netflix Sidecar 的灵感来自 Netflix Prana，它包括了一个简单的 HTTP API 来获取指定服务所有实例的信息（例如主机和端口）。不仅如此，还可通过内嵌的 Zuul 来代理服务调用，该代理从 Eureka Server 中获取信息。非 JVM 微服务需要实现健康检查，以便 Sidecar 将它的状态上报给 Eureka Server，健康检查的形式如下：

```
{
  "status":"UP"
}
```

其中，status 用于描述微服务的状态，常见取值有 UP、DOWN、OUT_OF_SERVICE 以及 UNKNOWN 等。

下面来详细讨论如何使用 Sidecar 整合非 JVM 微服务。

8.11.1 编写 Node.js 微服务

先来编写一个非 JVM 微服务，以 Node.js 为例进行演示。

代码如下，记为 node-service。

```
var http = require('http');
var url = require("url");
var path = require('path');

// 创建server
var server = http.createServer(function(req, res) {
  // 获得请求的路径
  var pathname = url.parse(req.url).pathname;
  res.writeHead(200, { 'Content-Type' : 'application/json; charset=utf-8' });
  // 访问http://localhost:8060/，将会返回{"index":"欢迎来到首页"}
  if (pathname === '/') {
    res.end(JSON.stringify({ "index" : "欢迎来到首页" }));
  }
  // 访问http://localhost:8060/health，将会返回{"status":"UP"}
  else if (pathname === '/health.json') {
    res.end(JSON.stringify({ "status" : "UP" }));
  }
  // 其他情况返回404
  else {
    res.end("404");
  }
});
// 创建监听，并打印日志
server.listen(8060, function() {
  console.log('listening on localhost:8060');
});
```

这是一个非常简单的 Node.js 服务。笔者已在代码注释中详细描述，不再赘述。

> **测试**
>
> 1. 使用`node node-service.js`命令即可启动该服务。
> 2. 访问 http://localhost:8060/health.json，将会返回如下内容。

```
{
    "status": "UP"
}
```

3. 访问 http://localhost:8060/，将会返回如下内容。

```
{
    "index": "欢迎来到首页"
}
```

4. 访问其他端点将会返回 404。

8.11.2 编写 Sidecar

本节来编写 Sidecar 微服务，并使用 Sidecar 整合非 JVM 微服务。

1. 创建一个 Maven 工程，ArtifactId 是`microservice-sidecar`，并为项目添加以下依赖。

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-zuul</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-netflix-sidecar</artifactId>
</dependency>
```

2. 在启动类上添加`@EnableSidecar`注解，声明这是一个 Sidecar。

```
@SpringBootApplication
@EnableSidecar
public class SidecarApplication {
  public static void main(String[] args) {
    SpringApplication.run(SidecarApplication.class, args);
  }
}
```

`@EnableSidecar`是一个组合注解，它整合了三个注解，分别是：`@EnableCircuitBreaker`、`@EnableDiscoveryClient`和`@EnableZuulProxy`。

3. 在项目的配置文件 application.yml 中添加如下内容：

```
server:
  port: 8070
spring:
  application:
    name: microservice-sidecar-node-service
eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/
  instance:
    prefer-ip-address: true
sidecar:
  port: 8060                                  # Node.js微服务的端口
  health-uri: http://localhost:8060/health.json   # Node.js微服务的健康检查URL
```

由配置可知，现在已经把 Sidecar 注册到了 Eureka Server 上，并用 sidecar.port 属性指定了非 JVM 微服务所监听的端口，用 sidecar.health-uri 属性指定了非 JVM 微服务的健康检查 URL。

这样一个 Sidecar 就编写完成了。

测试 1：非 JVM 微服务访问 JVM 微服务

1. 启动 microservice-discovery-eureka。
2. 启动 microservice-provider-user。
3. 启动 node-service。
4. 启动 microservice-sidecar。
5. 访问*http://localhost:8070/microservice-provider-user/1*，可获得如下结果。

```
{
  "id": 1,
  "username": "account1",
  "name": "张三",
  "age": 20,
  "balance": 100
}
```

因此，非 JVM 微服务可通过这种方式调用注册在 Eureka Server 上的 JVM 微服务。

测试 2：JVM 微服务调用非 JVM 微服务的接口

使用 Sidecar 微服务的 serviceId，即可调用非 JVM 微服务的接口。以下是一个使用 Ribbon 请求 node-service 的示例：

```
@GetMapping("/test")
public String findById() {
  // 将会请求到：http://localhost:8060/，返回结果：{"index":"欢迎来到首页"}
  return this.restTemplate.getForObject("http://microservice-sidecar-node-
      service/", String.class);
}
```

详见本书配套代码中的 microservice-sidecar-client-ribbon 项目。

8.11.3 Sidecar 的端点

Sidecar 提供了一些端点，这些端点有助于管理 Sidecar。

1. /

   该端点返回一个测试页面，该页面展示 Sidecar 的常用端点。

2. /hosts/{serviceId}

   该端点返回 DiscoveryClient.getInstances(serviceId)，即指定微服务在 Eureka 上的实例列表。

3. /ping

   该端点返回“OK”字符串。

4. /{serviceId}

   由于 Sidecar 整合了 Zuul，因此可使用该端点，将请求转发到 serviceId 所对应的微服务。

相关代码可详见：org.springframework.cloud.netflix.sidecar.SidecarController。

8.11.4 Sidecar 与 Node.js 微服务分离部署

前文是将 Sidecar 与非 JVM 微服务部署在同一台主机上。现实中，常常会将 Sidecar 与 JVM 微服务分离部署，例如部署在不同的主机或者容器中。此时应该如何配置呢？

对这个问题，笔者做了一点总结。

方法一：

```
eureka:
  instance:
    hostname: 非JVM微服务的hostname
```

方法二：

对于 Spring Cloud Netflix 1.3.0 及以后的版本，除方法一外，还可通过以下属性实现分离部署。

```
sidecar:
  hostname: 非JVM微服务的hostname
  ip-address: 非JVM微服务的IP地址
```

> 读者可参考该 Issue 辅助理解：*https://github.com/spring-cloud/spring-cloud-netflix/issues/981*。

8.11.5 Sidecar 原理分析

本节来探讨 Sidecar 的原理。

1. 访问 Eureka Server 的路径：*http://localhost:8761/eureka/apps/microservice-sidecar-node-service*，可获得类似于如下的结果。

```
<application>
  <name>MICROSERVICE-SIDECAR-NODE-SERVICE</name>
  <instance>
    <instanceId>itmuch:microservice-sidecar-node-service:8070</instanceId>
    <hostName>192.168.0.59</hostName>
    <app>MICROSERVICE-SIDECAR-NODE-SERVICE</app>
    <ipAddr>192.168.0.59</ipAddr>
    <status>UP</status>
    <overriddenstatus>UNKNOWN</overriddenstatus>
    <port enabled="true">8060</port>
    <securePort enabled="false">443</securePort>
    <countryId>1</countryId>
    <dataCenterInfo class="com.netflix.appinfo.InstanceInfo$
        DefaultDataCenterInfo">
```

```
      <name>MyOwn</name>
    </dataCenterInfo>
    <leaseInfo>
      <renewalIntervalInSecs>30</renewalIntervalInSecs>
      <durationInSecs>90</durationInSecs>
      <registrationTimestamp>1481963830415</registrationTimestamp>
      <lastRenewalTimestamp>1481963830415</lastRenewalTimestamp>
      <evictionTimestamp>0</evictionTimestamp>
      <serviceUpTimestamp>1481963829086</serviceUpTimestamp>
    </leaseInfo>
    <metadata class="java.util.Collections$EmptyMap"/>
    <homePageUrl>http://itmuch:8060/</homePageUrl>
    <statusPageUrl>http://itmuch:8070/info</statusPageUrl>
    <healthCheckUrl>http://itmuch:8070/health</healthCheckUrl>
    <vipAddress>microservice-sidecar-node-service</vipAddress>
    <secureVipAddress>microservice-sidecar-node-service</secureVipAddress>
    <isCoordinatingDiscoveryServer>false</isCoordinatingDiscoveryServer>
    <lastUpdatedTimestamp>1481963830415</lastUpdatedTimestamp>
    <lastDirtyTimestamp>1481963825748</lastDirtyTimestamp>
    <actionType>ADDED</actionType>
  </instance>
</application>
```

由结果可知，注册到 Eureka Server 上的端口是 8060，homePageUrl 是*http://itmuch:8060/*，恰恰是 node-service 的端口和首页。因此，注册到 Eureka Server 上的微服务可使用 microservice-sidecar-node-service 这个名称请求 node-service 的接口。

2. 由于@EnableSidecar整合了注解@EnableZuulProxy，可尝试访问 Sidecar 的/routes 端点：*http://localhost:8070/routes*，获得类似如下的结果。

```
{
  /microservice-provider-user/**: "microservice-provider-user"
}
```

因此，非 JVM 微服务可通过 Sidecar 请求其他注册在 Eureka Server 的微服务。

3. 可尝试将 node-service 多次启停，并观察 Sidecar 的/health 端点。Sidecar 会获取 node-service 的健康状态，并将该状态传播到 Eureka Server。使用这种方式，Eureka Server 就能感知到非 JVM 微服务的健康状态。

相关代码如下：

- `org.springframework.cloud.netflix.sidecar.LocalApplicationHealthCheckHandler.getStatus(InstanceStatus)`
- `org.springframework.cloud.netflix.sidecar.LocalApplicationHealthIndicator.doHealthCheck(Builder)`

8.12 使用 Zuul 聚合微服务

许多场景下，外部请求需要查询 Zuul 后端的多个微服务。举个例子，一个电影售票手机 APP，在购票订单页上，既需要查询“电影微服务”获得电影相关信息，又需要查询“用户微服务”获得当前用户的信息。如果让手机端直接请求各个微服务（即使使用 Zuul 进行转发），那么网络开销、流量耗费、耗费时长可能都无法令我们满意。那么对于这种场景，可使用 Zuul 聚合微服务请求——手机 APP 只需发送一个请求给 Zuul，由 Zuul 请求用户微服务以及电影微服务，并组织好数据给手机 APP。

使用这种方式，手机端只须发送一次请求即可，简化了客户端侧的开发；不仅如此，由于 Zuul、用户微服务、电影微服务一般都在同一个局域网中，因此速度会非常快，效率会非常高。

下面围绕以上场景，来编写代码示例。在本例中，使用 RxJava 结合 Zuul 来实现微服务请求的聚合。

1. 复制项目`microservice-gateway-zuul`，将 ArtifactId 修改为`microservice-gateway-zuul-aggregation`。
2. 修改启动类 ZuulApplication：

```
@SpringBootApplication
@EnableZuulProxy
public class ZuulApplication {
  public static void main(String[] args) {
    SpringApplication.run(ZuulApplication.class, args);
  }

  @Bean
  @LoadBalanced
  public RestTemplate restTemplate() {
```

```
    return new RestTemplate();
  }
}
```

3. 创建实体类：

```
public class User {
  private Long id;
  private String username;
  private String name;
  private Integer age;
  private BigDecimal balance;
  // getters and setters ...
}
```

4. 创建 Java 类，名为 AggregationService：

```
@Service
public class AggregationService {
  @Autowired
  private RestTemplate restTemplate;

  @HystrixCommand(fallbackMethod = "fallback")
  public Observable<User> getUserById(Long id) {
    // 创建一个被观察者
    return Observable.create(observer -> {
      // 请求用户微服务的/{id}端点
      User user = restTemplate.getForObject("http://microservice-provider-user/{
        id}", User.class, id);
      observer.onNext(user);
      observer.onCompleted();
    });
  }

  @HystrixCommand(fallbackMethod = "fallback")
  public Observable<User> getMovieUserByUserId(Long id) {
    return Observable.create(observer -> {
      // 请求电影微服务的/user/{id}端点
      User movieUser = restTemplate.getForObject("http://microservice-consumer-
          movie/user/{id}", User.class, id);
      observer.onNext(movieUser);
      observer.onCompleted();
```

```
    });
  }

  public User fallback(Long id) {
    User user = new User();
    user.setId(-1L);
    return user;
  }
}
```

5. 创建 Controller，在 Controller 中聚合多个请求。

```
@RestController
public class AggregationController {
  public static final Logger LOGGER = LoggerFactory.getLogger(ZuulApplication.
      class);

  @Autowired
  private AggregationService aggregationService;

  @GetMapping("/aggregate/{id}")
  public DeferredResult<HashMap<String, User>> aggregate(@PathVariable Long id)
      {
    Observable<HashMap<String, User>> result = this.aggregateObservable(id);
    return this.toDeferredResult(result);
  }

  public Observable<HashMap<String, User>> aggregateObservable(Long id) {
    // 合并两个或者多个Observables发射出的数据项，根据指定的函数变换它们
    return Observable.zip(
        this.aggregationService.getUserById(id),
        this.aggregationService.getMovieUserByUserId(id),
        (user, movieUser) -> {
          HashMap<String, User> map = Maps.newHashMap();
          map.put("user", user);
          map.put("movieUser", movieUser);
          return map;
        }
    );
  }
```

```
  public DeferredResult<HashMap<String, User>> toDeferredResult(Observable<
      HashMap<String, User>> details) {
    DeferredResult<HashMap<String, User>> result = new DeferredResult<>();
    // 订阅
    details.subscribe(new Observer<HashMap<String, User>>() {
      @Override
      public void onCompleted() {
        LOGGER.info("完成...");
      }

      @Override
      public void onError(Throwable throwable) {
        LOGGER.error("发生错误...", throwable);
      }

      @Override
      public void onNext(HashMap<String, User> movieDetails) {
        result.setResult(movieDetails);
      }
    });
    return result;
  }
}
```

这样，代码就编写完成了。相信熟悉 RxJava 的读者朋友们能非常轻松地阅读本段代码的含义，对于不了解的 RxJava 的读者朋友们，建议花一点时间入门 RxJava。

测试一：微服务聚合测试

1. 启动项目 microservice-discovery-eureka；
2. 启动项目 microservice-provider-user；
3. 启动项目 microservice-consumer-movie；
4. 启动项目 microservice-gateway-zuul-aggregation；
5. 访问*http://localhost:8040/aggregate/1* ，可获得如下结果：

```
{
  "movieUser": {
    "id": 1,
```

```
    "username": "account1",
    "name": "张三",
    "age": 20,
    "balance": 100.00
  },
  "user": {
    "id": 1,
    "username": "account1",
    "name": "张三",
    "age": 20,
    "balance": 100.00
  }
}
```

说明已成功用 Zuul 聚合了用户微服务以及电影微服务的 RESTful API。

测试二：Hystrix 容错测试

1. 在测试一的基础上，停止项目 microservice-provider-user 以及 microservice-consumer-movie；
2. 访问*http://localhost:8040/aggregate/1*，可获得如下结果：

```
{
  "movieUser": {
    "id": -1,
    "username": null,
    "name": null,
    "age": null,
    "balance": null
  },
  "user": {
    "id": -1,
    "username": null,
    "name": null,
    "age": null,
    "balance": null
  }
}
```

说明 fallback 方法正常被触发，能够正常回退。

9 使用 Spring Cloud Config 统一管理微服务配置

9.1 为什么要统一管理微服务配置

对于传统的单体应用，常使用配置文件管理所有配置。例如一个 Spring Boot 开发的单体应用，可将配置内容放在 application.yml 文件中。如果需要切换环境，可设置多个 Profile，并在启动应用时指定 spring.profiles.active={profile}。在本书 4.6 节中使用的也是这种方式。当然也可借助 Maven 的 Profile 实现环境切换。

然而，在微服务架构中，微服务的配置管理一般有以下需求：

- 集中管理配置。一个使用微服务架构的应用系统可能会包含成百上千个微服务，因此集中管理配置是非常有必要的。
- 不同环境不同配置。例如，数据源配置在不同的环境（开发、测试、预发布、生产等）中是不同的。
- 运行期间可动态调整。例如，可根据各个微服务的负载情况，动态调整数据源连接池大小或熔断阈值，并且在调整配置时不停止微服务。

- 配置修改后可自动更新。如配置内容发生变化，微服务能够自动更新配置。

综上所述，对于微服务架构而言，一个通用的配置管理机制是必不可少的，常见做法是使用配置服务器管理配置。

9.2 Spring Cloud Config 简介

Spring Cloud Config 为分布式系统外部化配置提供了服务器端和客户端的支持，它包括 Config Server 和 Config Client 两部分。由于 Config Server 和 Config Client 都实现了对 Spring Environment 和 PropertySource 抽象的映射，因此，Spring Cloud Config 非常适合 Spring 应用程序，当然也可与任何其他语言编写的应用程序配合使用。

Config Server 是一个可横向扩展、集中式的配置服务器，它用于集中管理应用程序各个环境下的配置，默认使用 Git 存储配置内容（也可使用 Subversion、本地文件系统或 Vault 存储配置，限于篇幅，本书不做讨论），因此可以很方便地实现对配置的版本控制与内容审计。

Config Client 是 Config Server 的客户端，用于操作存储在 Config Server 中的配置属性。如图 9-1 所示，所有的微服务都指向 Config Server。各个微服务在启动时，会请求 Config Server 以获取所需要的配置属性，然后缓存这些属性以提高性能。

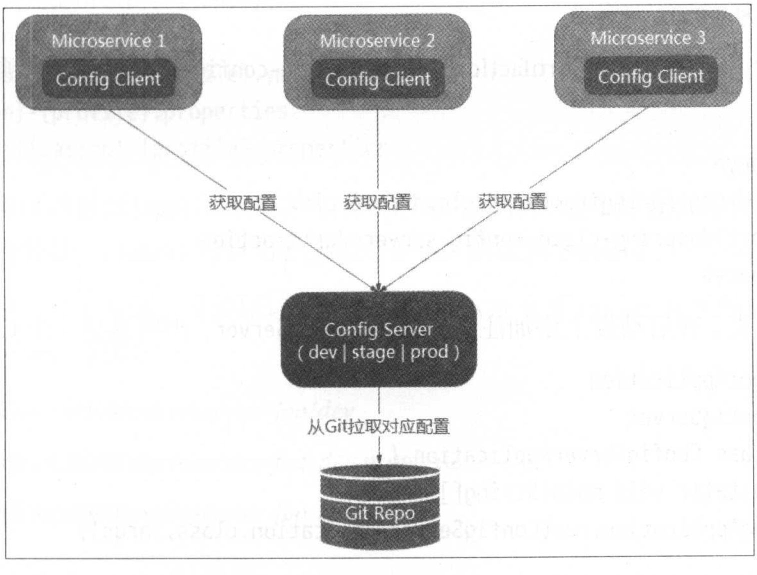


图 9-1 Spring Cloud Config 架构图

> Spring Cloud Config 的 GitHub 地址：*https://github.com/spring-cloud/spring-cloud-config*。

9.3 编写 Config Server

本节来编写一个 Config Server。在本例中，使用 Git 作为 Config Server 的后端存储。

1. 在 Git 仓库*https://git.oschina.net/itmuch/spring-cloud-config-repo*中新建几个配置文件，例如：

   ```
   microservice-foo.properties
   microservice-foo-dev.properties
   microservice-foo-test.properties
   microservice-foo-production.properties
   ```

   内容分别是：

   ```
   profile=default-1.0
   profile=dev-1.0
   profile=test-1.0
   profile=production-1.0
   ```

   为了测试版本控制，为该 Git 仓库创建 config-label-v2.0 分支，并将各个配置文件中的 1.0 改为 2.0。

2. 创建一个 Maven 工程，ArtifactId 是`microservice-config-server`，并为项目添加以下依赖。

   ```
   <dependency>
     <groupId>org.springframework.cloud</groupId>
     <artifactId>spring-cloud-config-server</artifactId>
   </dependency>
   ```

3. 编写启动类，在启动类上添加注解`@EnableConfigServer`，声明这是一个 Config Server。

   ```
   @SpringBootApplication
   @EnableConfigServer
   public class ConfigServerApplication {
     public static void main(String[] args) {
       SpringApplication.run(ConfigServerApplication.class, args);
     }
   }
   ```

4. 编写配置文件 application.yml，并在其中添加以下内容。

```
server:
  port: 8080
spring:
  application:
    name: microservice-config-server
  cloud:
    config:
      server:
        git:
          # 配置Git仓库的地址
          uri: https://git.oschina.net/itmuch/spring-cloud-config-repo
          # Git仓库的账号
          username:
          # Git仓库的密码
          password:
```

这样，一个 Config Server 就完成了。

Config Server 的端点

可以使用 Config Server 的端点获取配置文件的内容。端点与配置文件的映射规则如下：

```
/{application}/{profile}[/{label}]
/{application}-{profile}.yml
/{label}/{application}-{profile}.yml
/{application}-{profile}.properties
/{label}/{application}-{profile}.properties
```

以上端点都可以映射到`{application}-{profile}.properties`这个配置文件，`{application}`表示微服务的名称，`{label}`对应 Git 仓库的分支，默认是 master。

按照以上规则，对于本例，可使用以下 URL 访问到 Git 仓库 master 分支的 microservice-foo-dev.properties，例如：

- *http://localhost:8080/microservice-foo/dev*
- *http://localhost:8080/microservice-foo-dev.properties*
- *http://localhost:8080/microservice-foo-dev.yml*

测试

1. 访问*http://localhost:8080/microservice-foo/dev*，可获得如下结果。

```
{
    "name": "microservice-foo",
    "profiles": [
        "dev"
    ],
    "label": "master",
    "version": "6791f5acf796efd14be92ec2b4fc31779ad97d46",
    "state": null,
    "propertySources": [
        {
            "name": "https://git.oschina.net/itmuch/spring-cloud-config-repo
                /microservice-foo-dev.properties",
            "source": {
                "profile": "dev-1.0"
            }
        },
        {
            "name": "https://git.oschina.net/itmuch/spring-cloud-config-repo
                /microservice-foo.properties",
            "source": {
                "profile": "default-1.0"
            }
        }
    ]
}
```

从结果可以直观地看到应用名称、项目 profile、Git label、Git version、配置文件 URL、配置详情等信息。

2. 访问*http://localhost:8080/microservice-foo-dev.properties*，返回配置文件中的属性：

```
profile: dev-1.0
```

3. 访问*http://localhost:8080/config-label-v2.0/microservice-foo-dev.properties*，可获得如下结果：

```
profile: dev-2.0
```

说明获得了 Git 仓库 config-label-v2.0 分支中的配置信息。

至此，已成功构建了 Config Server，并通过构造 URL 的方式，获取了 Git 仓库中的配置信息。

> 需要注意的是，访问*http://localhost:8080/microservice-foo/dev*，结果中类似*https://git.oschina.net/itmuch/spring-cloud-config-repo/microservice-foo-dev.properties* 的 URL 并不能访问。这是正常的，因为它并不代表配置文件的实际 URL 路径，而只是一个标识。有兴趣的读者可前往下列页面进行拓展阅读：*https://github.com/spring-cloud/spring-cloud-config/issues/571*。

9.4 编写 Config Client

前文已经构建了一个 Config Server，并使用 Config Server 端点获取配置内容。本节来讨论 Spring Cloud 微服务如何获取配置信息。

下面来编写一个微服务，该微服务整合了 Config Client。

1. 创建一个 Maven 工程，ArtifactId 是`microservice-config-client`，并为项目添加以下依赖。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

2. 创建一个基本的 Spring Boot 启动类。

```
@SpringBootApplication
public class ConfigClientApplication {
```

```
  public static void main(String[] args) {
    SpringApplication.run(ConfigClientApplication.class, args);
  }
}
```

3. 编写配置文件 application.yml，并在其中添加如下内容。

```
server:
  port: 8081
```

4. 创建配置文件 bootstrap.yml，并在其中添加如下内容。

```
spring:
  application:
    # 对应config server所获取的配置文件的{application}
    name: microservice-foo
  cloud:
    config:
      uri: http://localhost:8080/
      # profile对应config server所获取的配置文件中的{profile}
      profile: dev
      # 指定Git仓库的分支，对应config server所获取的配置文件的{label}
      label: master
```

其中：

spring.application.name：对应 Config Server 所获取的配置文件中的 {application}。

spring.cloud.config.uri：指定 Config Server 的地址，默认是*http://localhost:8888*。

spring.cloud.config.profile：profile 对应 Config Server 所获取的配置文件中的 {profile}。

spring.cloud.config.label：指定 Git 仓库的分支，对应 Config Server 所获取配置文件的 {label}。

值得注意的是，以上属性应配置在 bootstrap.yml，而不是 application.yml 中。如果配置在 application.yml 中，该部分配置就不能正常工作。例如，Config Client 会连接 spring.cloud.config.uri 的默认值 *http://localhost:8888*，而并非配置的*http://localhost:8080/*。

Spring Cloud 有一个“引导上下文”的概念，这是主应用程序的父上下文。引导上下文负责从配置服务器加载配置属性，以及解密外部配置文件中的属性。和主应用程序加载`application.*` (yml 或 properties) 中的属性不同，引导上下文加载`bootstrap.*`中的属性。配置在`bootstrap.*` 中的属性有更高的优先级，因此默认情况下它们不能被本地配置覆盖。

如需禁用引导过程，可设置`spring.cloud.bootstrap.enabled=false`。

5. 编写 Controller。

```
@RestController
public class ConfigClientController {
  @Value("${profile}")
  private String profile;

  @GetMapping("/profile")
  public String hello() {
    return this.profile;
  }
}
```

在 Controller 中，通过注解`@Value("${profile}")`，绑定 Git 仓库配置文件中的 profile 属性。

测试

1. 启动 microservice-config-server。
2. 启动 microservice-config-client。
3. 访问*http://localhost:8081/profile*，可获得如下的结果。

```
dev-1.0
```

说明 Config Client 能够正常通过 Config Server 获得 Git 仓库中对应环境的配置。

Spring Cloud 引导上下文详解：*http://cloud.spring.io/spring-cloud-static/Camden.SR4/#_the_bootstrap_application_context*。

9.5 Config Server 的 Git 仓库配置详解

前文中，使用`spring.cloud.config.server.git.uri`指定了一个 Git 仓库，事实上，该属性非常灵活。本节来详细讨论 Config Server 的 Git 仓库配置。

1. 占位符支持

Config Server 的占位符支持 {application}、{profile} 和 {label}。

示例：

```yaml
server:
  port: 8080
spring:
  application:
    name: microservice-config-server
  cloud:
    config:
      server:
        git:
          uri: https://git.oschina.net/itmuch/{application}
          username:
          password:
```

使用这种方式，即可轻松支持一个应用对应一个 Git 仓库。同理，也可支持一个 profile 对应一个 Git 仓库。

2. 模式匹配

模式匹配指的是带有通配符的 {application}/{profile} 名称的列表。如果 {application}/{profile} 不匹配任何模式，它将会使用`spring.cloud.config.server.git.uri`定义的 URI。

```yaml
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          repos:
            simple: https://github.com/simple/config-repo
            special:
              pattern: special*/dev*,*special*/dev*
              uri: https://github.com/special/config-repo
            local:
              pattern: local*
              uri: file:/home/configsvc/config-repo
```

该例中，对于 simple 仓库，它只匹配所有配置文件中名为 simple 的应用程序。local 仓库则匹配所有配置文件中以 local 开头的所有应用程序的名称。

3. 搜索目录

很多场景下，可能把配置文件放在了 Git 仓库子目录中，此时可以使用 search-path 指定，search-path 同样支持占位符。

```
spring:
  cloud:
    config:
      server:
        git:
          uri: http://git.oschina.net/itmuch/spring-cloud-config-repo
          search-paths: foo,bar*
```

这样，Config Server 就会在 Git 仓库根目录、foo 子目录，以及所有以 bar 开始的子目录中查找配置文件。

4. 启动时加载配置文件

默认情况下，在配置被首次请求时，Config Server 才会 clone Git 仓库。也可让 Config Server 在启动时就 clone Git 仓库，例如:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          repos:
            team-a:
                pattern:  microservice-*
                clone-on-start: true
                uri: http://git.oschina.net/itmuch/spring-cloud-config-repo
```

将属性`spring.cloud.config.server.git.repos.*.clone-on-start`设为 true，即可让 Config Server 启动时 clone 指定 Git 仓库。

当然，也可使用`spring.cloud.config.server.git.clone-on-start = true`进行全局配置。

配置 clone-on-start = true，可帮助 Config Server 启动时快速识别错误的配置源（例如无效的 Git 仓库）。

将以下包的日志级别设为DEBUG，就可打印Config Server请求Git仓库的细节。可以通过这种方式，更好地理解Config Server的Git仓库配置，同时也便于快速定位问题。

```
logging:
  level:
    org.springframework.cloud: DEBUG
    org.springframework.boot: DEBUG
```

9.6 Config Server的健康状况指示器

Config Server自带了一个健康状况指示器，用于检查所配置的EnvironmentRepository是否正常工作。可使用Config Server的/health端点查询当前健康状态。默认情况下，健康指示器向EnvironmentRepository请求的{application}是app，{profile}和{label}是对应EnvironmentRepository实现的默认值。对于Git，{profile}是default，{label}是master。

同样也可以自定义健康状况指示器的配置，从而检查更多的{application}、自定义的{profile}以及自定义的{label}，例如：

```
server:
  port: 8080
spring:
  application:
    name: microservice-config-server
  cloud:
    config:
      server:
        git:
          # 配置Git仓库的地址
          uri: https://git.oschina.net/itmuch/spring-cloud-config-repo/
          # Git仓库的账号
          username:
          # Git仓库的密码
          password:
        health:
          repositories:
            a-foo:
              label: config-label-v2.0
```

```
    name: microservice-foo
    profiles: dev
```

详见本书配套代码中的 microservice-config-server-health 项目。

如需禁用健康状况指示器，可设置`spring.cloud.config.server.health.enabled=false`。

9.7 配置内容的加解密

前文是在 Git 仓库中明文存储配置属性的。很多场景下，对于某些敏感的配置内容（例如数据库账号、密码等），应当加密存储。

Config Server 为配置内容的加密与解密提供了支持。

9.7.1 安装 JCE

Config Server 的加解密功能依赖 Java Cryptography Extension（JCE）。

Java 8 JCE 的地址是：*http://www.oracle.com/technetwork/java/javase/downloads/jce8-download-2133166.html*。

下载 JCE 并解压，按照其中的 README.txt 的说明安装。JCE 的安装非常简单，其实就是将 JDK/jre/lib/security 目录中的两个 jar 文件替换为压缩包中的 jar 文件。

其他 Java 版本的 JCE 地址，在 Spring Cloud 文档中都有提及，详见：*http://cloud.spring.io/spring-cloud-static/Camden.SR4/#_cloud_native_applications*。

9.7.2 Config Server 的加解密端点

Config Server 提供了加密与解密的端点，分别是`/encrypt` 与 `/decrypt`。可使用以下代码来加密明文：

```
curl $CONFIG_SERVER_URL/encrypt -d 想要加密的明文
```

使用以下代码来解密密文：

```
curl $CONFIG_SERVER_URL/decrypt -d 想要解密的密文
```

9.7.3 对称加密

1. 复制项目`microservice-config-server`，将 ArtifactId 修改为`microservice-config-server-encryption`。
2. 修改 application.yml，添加以下内容。

```
encrypt:
  key: foo  # 设置对称密钥
```

这样，代码就编写完成了。

> **测试**
>
> 1. 输入命令：
>
>    ```
>    curl http://localhost:8080/encrypt -d mysecret
>    ```
>
>   可返回 851a6effab6619f43157a714061f4602be0131b73b56b0451a7e268c880daea3，说明 mysecret 已被加密。
>
> 2. 输入命令：
>
>    ```
>    curl http://localhost:8080/decrypt -d
>    851a6effab6619f43157a714061f4602be0131b73b56b0451a7e268c880daea3
>    ```
>
>   可返回 mysecret，说明能够正常解密。

9.7.4 存储加密的内容

加密后的内容，可使用 {cipher} 密文的形式存储。

1. 准备一个配置文件，命名为 encryption.yml。

   ```
   spring:
     datasource:
       username: dbuser
       password: '{cipher}851a6effab6619f43157a714061f4602be0131b73b56b0451a7e268c8
   80daea3'
   ```

   并将其 push 到 Git 仓库 *https://git.oschina.net/itmuch/spring-cloud-config-repo*。此处需注意 spring.datasource.password 上的单引号不能少。如读者使用 properties 格式管理配置，则不能使用单引号，否则该值不会被解密，正确写法如下：

   ```
   spring.datasource.username=dbuser
   spring.datasource.password={cipher}851a6effab6619f43157a714061f4602be0131b73b56b
   0451a7e268c880daea3
   ```

2. 使用 *http://localhost:8080/encryption-default.yml* 可获得如下结果。

   ```
   profile: default
   spring:
     datasource:
   ```

```
password: mysecret
username: dbuser
```

说明 Config Server 能自动解密配置内容。

一些场景下，想要让 Config Server 直接返回密文本身，而并非解密后的内容，可设置`spring.cloud.config.server.encrypt.enabled=false`，这时可由 Config Client 自行解密。

9.7.5 非对称加密

前文讨论了对称加密的方式，Spring Cloud 同样支持非对称加密。

1. 复制项目`microservice-config-server`，将 ArtifactId 修改为`microservice-config-server-encryption-rsa`。
2. 执行以下命令，并按照提示操作，即可创建一个 Key Store。

```
keytool -genkeypair -alias mytestkey -keyalg RSA -dname "CN=Web Server,OU=Unit,
O=Organization,L=City,S=State,C=US" -keypass changeme -keystore server.jks
-storepass letmein
```

3. 将生成的 server.jks 文件复制到项目的 classpath 下。
4. 在 application.yml 中添加以下内容。

```
encrypt:
  keyStore:
    location: classpath:/server.jks # jks文件的路径
    password: letmein               # storepass
    alias: mytestkey                # alias
    secret: changeme                # keypass
```

这样，使用以下命令：

```
curl http://localhost:8080/encrypt -d mysecret
```

尝试加密时，就会得到类似以下的结果：

```
AQB38UyNckYzW64rvsaIhy0OV4MUmS7krdHrw+VLUdqXJ4ZVdZL8/ouwSOAYM+6MSjKvzmkaU8Iv2cQ5
MWhlZhCrm0f0d2ubc1MH96KBHTix9AroajeTiofPwPoBnWfBo9cC4PU1vD+rcvAvwvdR5q7rYbFc4yut
4uJZRzpAXGgf680kAtb6tEtLx7c4/35PEaGXFWd2m8gn21vzWdvhbP6cdC9YlburL0Rq/0H1G+uEX99Z
VIWJ0hVn4rplLWPMLUGA2ZVEyVRorIRX/2z5MU7cVPtJ6X1JZDpU4GVz8/3rD5BnbVFTGo6DfBrEzJn5
8Bzjl6aqo9ca/3j42RHOoQDOHXGqRX/843RbPdvMqTZd0rTOBHTUrVG9E15sCajiLkw=
```

相对于对称加密，非对称加密的安全性更高，但对称加密相对方便。读者可按照需求，自行选择加密方案。

9.8　使用/refresh 端点手动刷新配置

很多场景下，需要在运行期间动态调整配置。如果配置发生了修改，微服务要如何实现配置的刷新呢?

要想实现配置刷新，须对之前的代码进行一点改造。

1. 复制项目`microservice-config-client`，将 ArtifactId 修改为`microservice-config-client-refresh`。
2. 为项目添加`spring-boot-starter-actuator` 的依赖，该依赖包含了/refresh 端点，用于配置的刷新。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

3. 在 Controller 上添加注解`@RefreshScope`。添加`@RefreshScope`的类会在配置更改时得到特殊的处理。

```
@RestController
@RefreshScope
public class ConfigClientController {
  @Value("${profile}")
  private String profile;

  @GetMapping("/profile")
  public String hello() {
    return this.profile;
  }
}
```

这样就完成了代码改造。

测试

1. 启动 microservice-config-server。
2. 启动 microservice-config-client-refresh。
3. 访问*http://localhost:8081/profile*，获得结果：dev-1.0。
4. 修改 Git 仓库中 microservice-foo-dev.properties 文件内容为 profile=dev-1.0-change。

5. 重新访问*http://localhost:8081/profile*，发现结果依然是 dev-1.0，说明配置尚未刷新。
6. 发送 POST 请求到*http://localhost:8081/refresh*，例如：

```
curl  -X POST http://localhost:8081/refresh
```

返回结果："profile"，表示 profile 这个配置属性已被刷新。

7. 再次访问*http://localhost:8081/profile*，返回 dev-1.0-change，说明配置已经刷新。

9.9 使用 Spring Cloud Bus 自动刷新配置

前文讨论了使用/refresh 端点手动刷新配置，但如果所有微服务节点的配置都需要手动去刷新，工作量可想而知。不仅如此，随着系统的不断扩张，会越来越难以维护。因此，实现配置的自动刷新是很有必要的，本节将使用 Spring Cloud Bus 实现配置的自动刷新。

9.9.1 Spring Cloud Bus 简介

Spring Cloud Bus 使用轻量级的消息代理（例如 RabbitMQ、Kafka 等）连接分布式系统的节点，这样就可以广播传播状态的更改（例如配置的更新）或者其他的管理指令。可将 Spring Cloud Bus 想象成一个分布式的 Spring Boot Actuator。使用 Spring Cloud Bus 后的架构如图 9-2 所示。

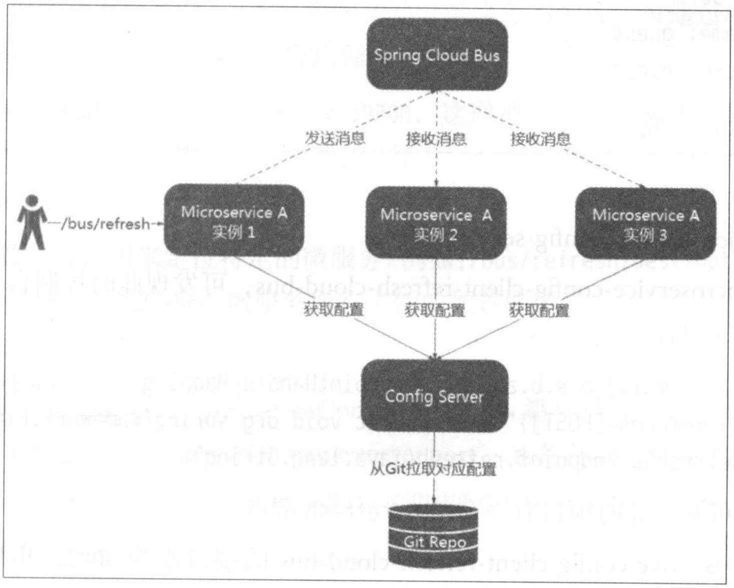


图 9-2 使用 Spring Cloud Bus 的架构图

由图可知，微服务 A 的所有实例都通过消息总线连接到了一起，每个实例都会订阅配置更新事件。当其中一个微服务节点的/bus/refresh 端点被请求时，该实例就会向消息总线发送一个配置更新事件，其他实例获得该事件后也会更新配置。

9.9.2 实现自动刷新

安装 RabbitMQ（详见 7.5.3.1 节）后，接下来为项目整合 Spring Cloud Bus 并实现自动刷新。

1. 复制项目`microservice-config-client-refresh`，将 ArtifactId 修改为 `microservice-config-client-refresh-cloud-bus`。
2. 为项目添加`spring-cloud-starter-bus-amqp`的依赖。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
```

3. 在 bootstrap.yml 中添加以下内容：

```
spring:
  rabbitmq:
    host: localhost
    port: 5672
    username: guest
    password: guest
```

这样代码就改造完成了。

测试

1. 启动 microservice-config-server。
2. 启动 microservice-config-client-refresh-cloud-bus，可发现此时控制台会输出类似于如下的内容。

```
[           main] o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/bus/refresh],methods=[POST]}" onto public void org.springframework.cloud.bus.endpoint.RefreshBusEndpoint.refresh(java.lang.String)
```

由结果可知，此时项目有一个`/bus/refresh` 端点。

3. 将 microservice-config-client-refresh-cloud-bus 的端口改为 8082，再启动一个节点。

4. 访问*http://localhost:8081/profile*，可获得结果：dev-1.0。
5. 将 Git 仓库中的 microservice-foo-dev.properties 文件内容修改为如下内容。

```
profile=dev-1.0-bus
```

6. 发送 POST 请求到其中一个 Config Client 实例的/bus/refresh 端点，例如：

```
curl -X POST http://localhost:8081/bus/refresh
```

7. 访问两个 Config Client 节点的/profile 端点，会发现两个节点都会返回`dev-1.0-bus`，说明配置内容已被刷新。

借助 Git 仓库的 WebHooks，就可轻松实现配置的自动刷新。如图 9-3 所示。

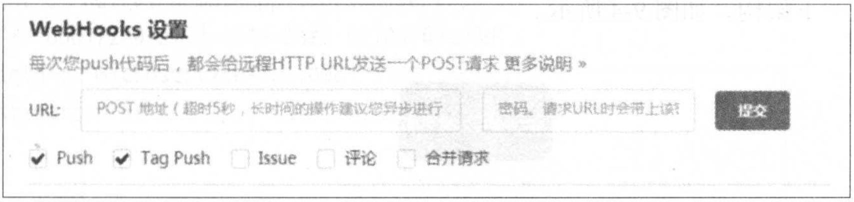


图 9-3 Git WebHooks 设置

9.9.3 局部刷新

某些场景下（例如灰度发布等），若只想刷新部分微服务的配置，可通过/bus/refresh 端点的 destination 参数来定位要刷新的应用程序。

例如：`/bus/refresh?destination=customers:9000`，这样消息总线上的微服务实例就会根据 destination 参数的值来判断是否需要刷新。其中，`customers:9000`指的是各个微服务的 ApplicationContext ID。

destination 参数也可以用来定位特定的微服务。例如：`/bus/refresh?destination=customers:**`，这样就可以触发 customers 微服务所有实例的配置刷新。

> 配置的局部刷新与 ApplicationContext ID 有关。默认情况下，ApplicationContext ID 是 spring.application.name:server.port。笔者在博客中对此有较为详细的分析，详见：*http://www.itmuch.com/spring-cloud-code-read/spring-cloud-code-read-spring-cloud-bus/*。

9.9.4 架构改进

在前面的示例中，通过请求某个微服务/bus/refresh 端点的方式来实现配置刷新，但这种方式并不优雅。原因如下：

1. 破坏了微服务的职责单一原则。业务微服务只应关注自身业务，不应承担配置刷新的职责。
2. 破坏了微服务各节点的对等性。
3. 有一定的局限性。例如，微服务在迁移时，网络地址常常会发生变化。此时如果想自动刷新配置，就不得不修改 WebHook 的配置。

不妨改进一下架构，如图 9-4 所示。

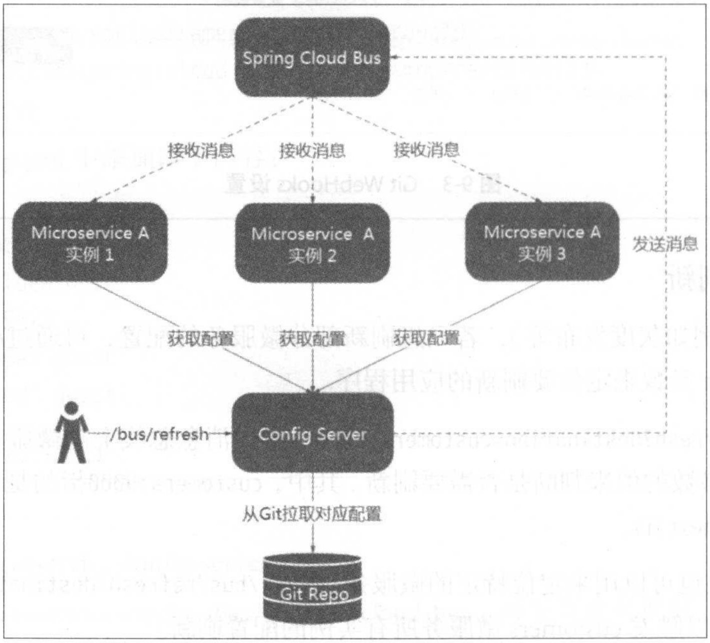


图 9-4 使用 Spring Cloud Bus 的架构图

如图 9-4 所示，**将 Config Server 也加入到消息总线中**，并使用 Config Server 的/bus/refresh 端点来实现配置的刷新。这样，各个微服务只需要关注自身的业务，而不再承担配置刷新的职责。

详见配套代码中的`microservice-config-server-refresh-cloud-bus`项目。

9.9.5 跟踪总线事件

一些场景下如果希望知道 Spring Cloud Bus 事件传播的细节，可以跟踪总线事件（RemoteApplicationEvent 的子类都是总线事件）。

想要跟踪总线事件非常简单，只须设置spring.cloud.bus.trace.enabled=true，这样在/bus/refresh 端点被请求后，访问/trace 端点就可获得类似如下的结果：

```
{
  "timestamp": 1481098786017,
  "info": {
    "signal": "spring.cloud.bus.ack",
    "event": "RefreshRemoteApplicationEvent",
    "id": "66d172e0-e770-4349-baf7-0210af62ea8d",
    "origin": "microservice-foo:8081",
    "destination": "**"
  }
},{
  "timestamp": 1481098779073,
  "info": {
    "signal": "spring.cloud.bus.sent",
    "type": "RefreshRemoteApplicationEvent",
    "id": "66d172e0-e770-4349-baf7-0210af62ea8d",
    "origin": "microservice-config-server:8080",
    "destination": "**:**"
  }
}...
```

这样就可清晰地知道事件的传播细节。

9.10 Spring Cloud Config 与 Eureka 配合使用

前文在微服务中指定了 Config Server 地址，这种方式无法利用服务发现组件的优势。本节将讨论 Config Server 注册到 Eureka Server 上时，如何使用 Spring Cloud Config。

1. 将 Config Server 和 Config Client 都注册到 Eureka Server 上。
2. Config Client 的 bootstrap.yml 可配置如下。

```
spring:
  application:
    # 对应config server所获取的配置文件的{application}
```

```
    name: microservice-foo
  cloud:
    config:
      profile: dev
      label: master
      discovery:
        # 表示使用服务发现组件中的Config Server，而不自己指定Config Server
        # 的uri，默认false
        enabled: true
        # 指定Config Server在服务发现中的serviceId，默认是configserver
        service-id: microservice-config-server-eureka
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

简要说明一下 bootstrap.yml 中的配置属性：

- `spring.cloud.config.discovery.enabled = true`：表示开启通过服务发现组件访问 Config Server 的功能；
- `spring.cloud.config.discovery.service-id`：指定 Config Server 在服务发现组件中的 serviceId。

详见本书配套代码中的 microservice-config-server-eureka 项目和 microservice-config-client-eureka 项目。

9.11 Spring Cloud Config 的用户认证

在前文的示例中，Config Server 是允许匿名访问的。为了防止配置内容的外泄，应该保护 Config Server 的安全。有多种方式做到这一点，例如通过物理网络安全，或者为 Config Server 添加用户认证等。

本节来为 Config Server 添加基于 HTTP Basic 的用户认证。

先来构建一个需要用户认证的 Config Server。

1. 复制项目`microservice-config-server`，将 ArtifactId 修改为`microservice-config-server-authenticating`。
2. 为项目添加以下依赖。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

3. 在 application.yml 中添加如下内容。

```
security:
  basic:
    enabled: true                    # 开启基于HTTP basic的认证
  user:
    name: user                       # 配置登录的账号是user
    password: password123            # 配置登录的密码是password123
```

这样就为 Config Server 添加了基于 HTTP basic 的认证。如果不设置这段内容，账号默认是 user，密码是一个随机值，该值会在启动时打印出来。

此时启动该 Config Server，将会看到如下的登录对话框，如图 9-5 所示。

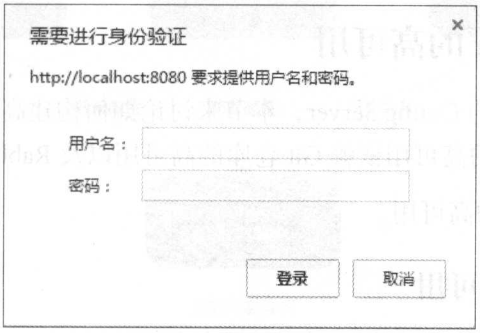


图 9-5 Config Server 登录界面

Config Client 连接需用户认证的 Config Server

Config Client 有两种方式使用需用户认证的 Config Server。

方式一：使用 curl 风格的 URL。示例：

```
spring:
  cloud:
    config:
      uri: http://user:password123@localhost:8080/
```

方法二：指定 Config Server 的账号与密码。示例：

```
spring:
  cloud:
    config:
      uri: http://localhost:8080/
      username: user
      password: password123
```

需要注意的是`spring.cloud.config.password`和`spring.cloud.config.username`的优先级较高，它们会覆盖 URL 中包含的账号与密码。

> 对于 Config Server 注册到 Eureka Server 的情况，可参考本书配套代码中的 microservice-config-server-eureka-authenticating 项目和 microservice-config-client-eureka-authenticating 项目。

9.12 Config Server 的高可用

前文构建的都是单节点的 Config Server，本节来讨论如何构建高可用的 Config Server 集群，包括 Config Server 的高可用依赖 Git 仓库的高可用以及 RabbitMQ 的高可用。

下面先来讨论 Git 仓库的高可用。

9.12.1 Git 仓库的高可用

由于配置内容都存储在 Git 仓库中，所以要想实现 Config Server 的高可用，必须有一个高可用的 Git 仓库。有两种方式可以实现 Git 仓库的高可用。

- 使用第三方 Git 仓库：这种方式非常简单，可使用例如 GitHub、BitBucket、git@osc、Coding 等提供的仓库托管服务，这些服务本身就已实现了高可用。
- 自建 Git 仓库管理系统：使用第三方服务的方式虽然省去了很多烦恼，但是很多场景下，倾向于自建 Git 仓库管理系统。此时就需要保证自建 Git 的高可用。

以 GitLab 为例，读者可参照官方文档搭建高可用的 GitLab：*https://about.gitlab.com/high-availability/*。

9.12.2 RabbitMQ 的高可用

还记得前文使用 Spring Cloud Bus 实现了配置的自动刷新吗？由于 Spring Cloud Bus 依赖 RabbitMQ（当然也可使用其他 MQ），所以 RabbitMQ 的高可用也是必不可少的。

搭建高可用 RabbitMQ 的资料详见：*https://www.rabbitmq.com/ha.html*。由于比较简单，笔者不做赘述。当然，也可使用云平台的提供的 RabbitMQ 服务。

9.12.3 Config Server 自身的高可用

本节来讨论如何实现 Config Server 自身的高可用。笔者分两种场景进行讨论。

Config Server 未注册到 Eureka Server 上

对于这种情况，Config Server 的高可用可借助一个负载均衡器来实现，如图 9-6 所示。

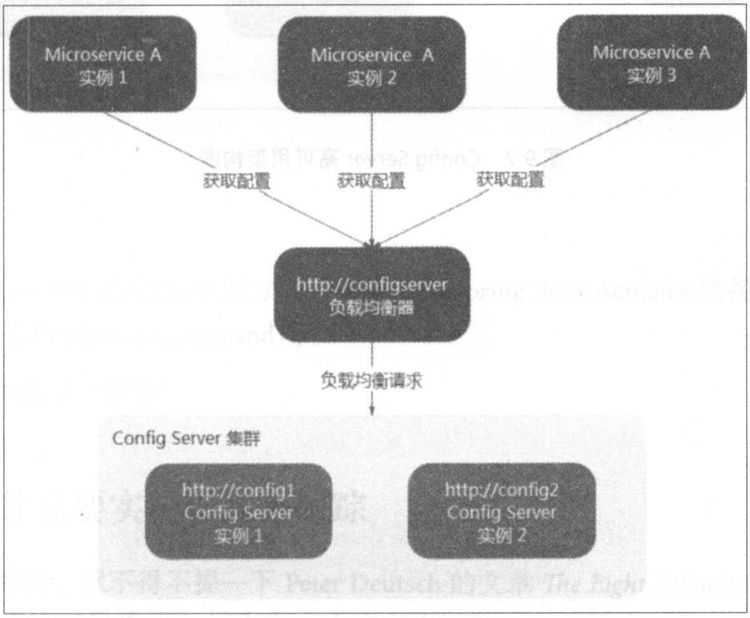


图 9-6 Config Server 高可用架构图

如图所示，各个微服务将请求发送到负载均衡器，负载均衡器将请求转发到其代理的其中一个 Config Server 节点。这样，就可以实现 Config Server 的高可用。

Config Server 注册到 Eureka Server 上

这种情况下，Config Server 的高可用相对简单，只须将多个 Config Server 节点注册到 Eureka Server 上，即可实现 Config Server 的高可用。架构如图 9-7 所示。

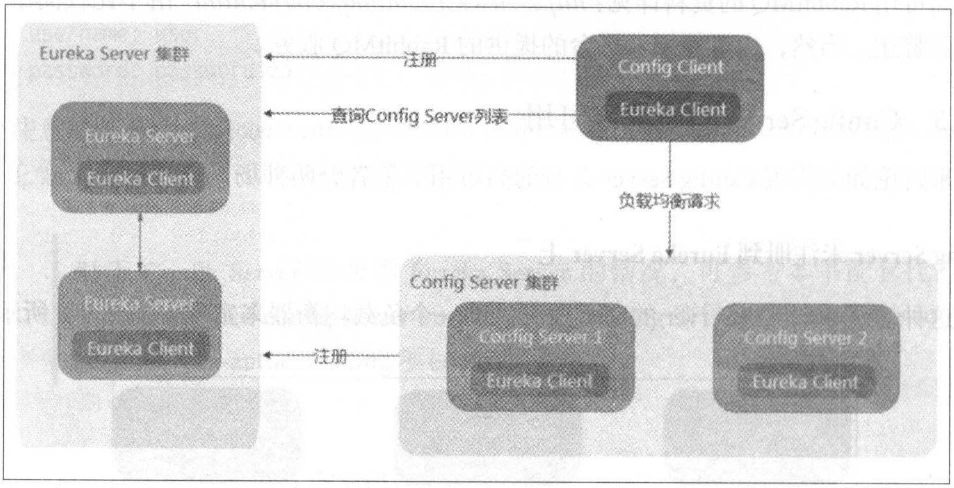


图 9-7 Config Server 高可用架构图

第10章 使用 Spring Cloud Sleuth 实现微服务跟踪

前文已经讲过几种监控微服务的方式，例如使用 Spring Boot Actuator 监控微服务实例，使用 Hystrix 监控 Hystrix Command 等。

本章来讨论微服务“跟踪”。

10.1 为什么要实现微服务跟踪

谈到微服务跟踪，就不得不提一下 Peter Deutsch 的文章 *The Eight Fallacies of Distributed Computing*（分布式计算的八大误区），内容大致如下：

- 网络可靠
- 延迟为零
- 带宽无限
- 网络绝对安全
- 网络拓扑不会改变

- 必须有一名管理员
- 传输成本为零
- 网络同质化（由 Java 之父 Golsing 补充）

从中可以看到，该文章很多点都在描述一个问题——网络问题。网络常常很脆弱，同时，网络资源也是有限的。

我们知道，微服务之间通过网络进行通信。如果能够跟踪每个请求，了解请求经过哪些微服务（从而了解信息是如何在服务之间流动）、请求耗费时间、网络延迟、业务逻辑耗费时间等指标，那么就能更好地分析系统瓶颈、解决系统问题。因此，微服务跟踪很有必要。

> *The Eight Fallacies of Distributed Computing*（分布式计算的八大误区）原文：*https://blogs.oracle.com/jag/resource/Fallacies.html*。

10.2 Spring Cloud Sleuth 简介

Spring Cloud Sleuth 为 Spring Cloud 提供了分布式跟踪的解决方案，它大量借用了 Google Dapper、Twitter Zipkin 和 Apache HTrace 的设计。

先来了解一下 Sleuth 的术语，Sleuth 借用了 Dapper 的术语。

- span（跨度）：基本工作单元。span 用一个 64 位的 id 唯一标识。除 ID 外，span 还包含其他数据，例如描述、时间戳事件、键值对的注解（标签），span ID、span 父ID 等。

  span 被启动和停止时，记录了时间信息。初始化 span 被称为“root span”，该 span 的 id 和 trace 的 ID 相等。
- trace（跟踪）：一组共享“root span”的 span 组成的树状结构称为 trace。trace 也用一个 64 位的 ID 唯一标识，trace 中的所有 span 都共享该 trace 的 ID。
- annotation（标注）：annotation 用来记录事件的存在，其中，核心 annotation 用来定义请求的开始和结束。
  - CS（Client Sent 客户端发送）：客户端发起一个请求，该 annotation 描述了 span 的开始。
  - SR（Server Received 服务器端接收）：服务器端获得请求并准备处理它。如果用 SR 减去 CS 时间戳，就能得到网络延迟。

- SS（Server Sent 服务器端发送）：该 annotation 表明完成请求处理（当响应发回客户端时）。如果用 SS 减去 SR 时间戳，就能得到服务器端处理请求所需的时间。
- CR（Client Received 客户端接收）：span 结束的标识。客户端成功接收到服务器端的响应。如果 CR 减去 CS 时间戳，就能得到从客户端发送请求到服务器响应的所需的时间。

图 10-1 详细描述了请求依次经过 SERVICE1—SERVICE2—SERVICE3—SERVICE4 时，span、trace、annotation 的变化。

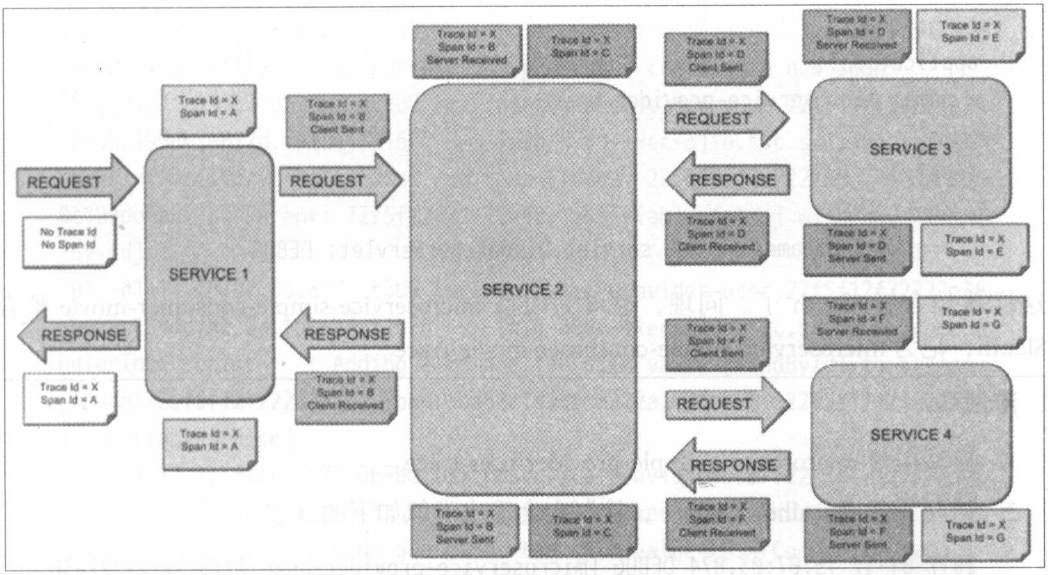


图 10-1　微服务追踪

> 读者如无法理解这些术语，也不必担心，在实战时会有直观的体验。有一定基础后再来看本节，就能深刻理解这些术语的含义。

> Dapper 论文：*https://research.google.com/pubs/pub36356.html*。

10.3 整合 Spring Cloud Sleuth

本节来为之前编写的项目整合 Sleuth。简单起见，使用 microservice-simple-provider-user 这个最原始的项目进行改造。

1. 复制项目microservice-simple-provider-user，将 ArtifactId 修改为microservice-simple-provider-user-trace。
2. 为项目添加以下依赖。

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
```

3. 修改 application.yml，添加以下内容：

```
spring:
  application:
    name: microservice-provider-user
logging:
  level:
    root: INFO
    org.springframework.web.servlet.DispatcherServlet: DEBUG
```

这样就整合好 Sleuth 了。同理，也可为项目 microservice-simple-consumer-movie 整合 Sleuth，记为 microservice-simple-consumer-movie-trace。

测试

1. 启动项目 microservice-simple-provider-user-trace。
2. 访问 http://localhost:8000/1，控制台会输出类似如下的日志。

```
2017-01-17 15:07:03.874 DEBUG [microservice-provider-user,22f5f12f22222a36
,22f5f12f22222a36,false] 11676 --- [nio-8000-exec-3] o.s.web.servlet.Dispa
tcherServlet         : DispatcherServlet with name 'dispatcherServlet' proc
essing GET request for [/1]
2017-01-17 15:07:03.877 DEBUG [microservice-provider-user,22f5f12f22222a36
,22f5f12f22222a36,false] 11676 --- [nio-8000-exec-3] o.s.web.servlet.Dispa
tcherServlet         : Last-Modified value for [/1] is: -1
2017-01-17 15:07:03.882 DEBUG [microservice-provider-user,22f5f12f22222a36
,ba88c199dbfa022a,false] 11676 --- [nio-8000-exec-3] o.s.web.servlet.Dispa
tcherServlet         : Null ModelAndView returned to DispatcherServlet with
 name 'dispatcherServlet': assuming HandlerAdapter completed request handl
ing
2017-01-17 15:07:03.883 DEBUG [microservice-provider-user,22f5f12f22222a36
,22f5f12f22222a36,false] 11676 --- [nio-8000-exec-3] o.s.web.servlet.Dispa
tcherServlet         : Successfully completed request
```

其中，22f5f12f22222a36 是 trace ID，ba88c199dbfa022a 等是 span ID。仔细分析日志，不难看出请求的具体过程。也可将日志如下设置：

```
logging:
  level:
    root: INFO
      org.springframework.cloud.sleuth: DEBUG
```

这样，就能了解 span 从创建到关闭的详细过程，例如：

```
2017-01-17 15:07:03.873 DEBUG [microservice-provider-user,22f5f12f22222a36
,22f5f12f22222a36,false] 11676 --- [nio-8000-exec-3] o.s.c.sleuth.instrume
nt.web.TraceFilter  : No parent span present - creating a new span
2017-01-17 15:07:03.877 DEBUG [microservice-provider-user,22f5f12f22222a36
,ba88c199dbfa022a,false] 11676 --- [nio-8000-exec-3] o.s.c.s.i.web.TraceHa
ndlerInterceptor    : Created new span [Trace: 22f5f12f22222a36, Span: ba8
8c199dbfa022a, Parent: 22f5f12f22222a36, exportable:false] with name [find
-by-id]
2017-01-17 15:07:03.877 DEBUG [microservice-provider-user,22f5f12f22222a36
,ba88c199dbfa022a,false] 11676 --- [nio-8000-exec-3] o.s.c.s.i.web.TraceHa
ndlerInterceptor    : Adding a method tag with value [findById] to a span
[Trace: 22f5f12f22222a36, Span: ba88c199dbfa022a, Parent: 22f5f12f22222a36
, exportable:false]
2017-01-17 15:07:03.877 DEBUG [microservice-provider-user,22f5f12f22222a36
,ba88c199dbfa022a,false] 11676 --- [nio-8000-exec-3] o.s.c.s.i.web.TraceHa
ndlerInterceptor    : Adding a class tag with value [UserController] to a
span [Trace: 22f5f12f22222a36, Span: ba88c199dbfa022a, Parent: 22f5f12f222
22a36, exportable:false]
2017-01-17 15:07:03.882 DEBUG [microservice-provider-user,22f5f12f22222a36
,ba88c199dbfa022a,false] 11676 --- [nio-8000-exec-3] o.s.c.s.i.web.TraceHa
ndlerInterceptor    : Closing span [Trace: 22f5f12f22222a36, Span: ba88c19
9dbfa022a, Parent: 22f5f12f22222a36, exportable:false]
2017-01-17 15:07:03.883 DEBUG [microservice-provider-user,22f5f12f22222a36
,22f5f12f22222a36,false] 11676 --- [nio-8000-exec-3] o.s.c.sleuth.instrume
nt.web.TraceFilter  : Closing the span [Trace: 22f5f12f22222a36, Span: 22f
5f12f22222a36, Parent: null, exportable:false] since the response was succ
essful
```

3. 启动项目 microservice-simple-consumer-movie-trace。
4. 访问*http://localhost:8010/user/1*会发现两个项目都会打印类似以上的日志。

10.4 Spring Cloud Sleuth 与 ELK 配合使用

ELK 是一款非常流行的日志分析系统，相信大家都不陌生。本节来为 Sleuth 整合 ELK。

1. 搭建 ELK。

   本书使用 ELK 5.1.2 进行测试。ELK 的搭建比较简单，读者可参考官方文档（*https://www.elastic.co/guide/index.html*）自行安装。

2. 复制项目`microservice-simple-provider-user-trace`，将 ArtifactId 修改为`microservice-simple-provider-user-trace-elk`。

3. 为项目添加以下依赖。

```
<dependency>
  <groupId>net.logstash.logback</groupId>
  <artifactId>logstash-logback-encoder</artifactId>
  <version>4.6</version>
</dependency>
```

4. 在项目的 src/main/resources 目录下新建文件 logback-spring.xml，在其中添加如下内容。

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <include resource="org/springframework/boot/logging/logback/defaults.xml" />

  <springProperty scope="context" name="springAppName" source="spring.
     application.name" />
  <!-- Example for logging into the build folder of your project -->
  <property name="LOG_FILE" value="${BUILD_FOLDER:-build}/${springAppName}" />

  <property name="CONSOLE_LOG_PATTERN"
    value="%clr(%d{yyyy-MM-dd HH:mm:ss.SSS}){faint} %clr(${LOG_LEVEL_PATTERN:-%5
        p}) %clr([${springAppName:-},%X{X-B3-TraceId:-},%X{X-B3-SpanId:-},%X{X-
        B3-ParentSpanId:-},%X{X-Span-Export:-}]){yellow} %clr(${PID:- }){magenta
        } %clr(---){faint} %clr([%15.15t]){faint} %clr(%-40.40logger{39}){cyan}
        %clr(:){faint} %m%n${LOG_EXCEPTION_CONVERSION_WORD:-%wEx}" />

  <!-- Appender to log to console -->
  <appender name="console" class="ch.qos.logback.core.ConsoleAppender">
    <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
      <!-- Minimum logging level to be presented in the console logs -->
      <level>DEBUG</level>
```

```
    </filter>
    <encoder>
      <pattern>${CONSOLE_LOG_PATTERN}</pattern>
      <charset>utf8</charset>
    </encoder>
  </appender>

  <!-- Appender to log to file -->
  <appender name="flatfile" class="ch.qos.logback.core.rolling.
      RollingFileAppender">
    <file>${LOG_FILE}</file>
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
      <fileNamePattern>${LOG_FILE}.%d{yyyy-MM-dd}.gz</fileNamePattern>
      <maxHistory>7</maxHistory>
    </rollingPolicy>
    <encoder>
      <pattern>${CONSOLE_LOG_PATTERN}</pattern>
      <charset>utf8</charset>
    </encoder>
  </appender>

  <!-- Appender to log to file in a JSON format -->
  <appender name="logstash" class="ch.qos.logback.core.rolling.
      RollingFileAppender">
    <file>${LOG_FILE}.json</file>
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
      <fileNamePattern>${LOG_FILE}.json.%d{yyyy-MM-dd}.gz</fileNamePattern>
      <maxHistory>7</maxHistory>
    </rollingPolicy>
    <encoder class="net.logstash.logback.encoder.
        LoggingEventCompositeJsonEncoder">
      <providers>
        <timestamp>
          <timeZone>UTC</timeZone>
        </timestamp>
        <pattern>
          <pattern>
            {
              "severity": "%level",
```

```
                "service": "${springAppName:-}",
                "trace": "%X{X-B3-TraceId:-}",
                "span": "%X{X-B3-SpanId:-}",
                "parent": "%X{X-B3-ParentSpanId:-}",
                "exportable": "%X{X-Span-Export:-}",
                "pid": "${PID:-}",
                "thread": "%thread",
                "class": "%logger{40}",
                "rest": "%message"
              }
            </pattern>
          </pattern>
        </providers>
      </encoder>
    </appender>

    <root level="INFO">
      <appender-ref ref="console" />
      <appender-ref ref="logstash" />
      <!--<appender-ref ref="flatfile"/> -->
    </root>
  </configuration>
```

5. 编写 bootstrap.yml，并将 application.yml 中的以下属性移动到 bootstrap.yml 中。

```
spring:
  application:
    name: microservice-provider-user
```

由于使用了自定义的 logback-spring.xml，并且该文件中含有变量（例如 springAppName），spring.application.name 属性必须设置在 bootstrap.yml 文件中，否则，logback-spring.xml 将无法正确读取属性。

6. 编写 Logstash 配置文件，命名为 logstash.conf，内容如下。

```
input {
  file {
    codec => json
    path => "/opt/build/*.json"  # 改成你项目打印的json日志文件。
  }
}
filter {
```

```
  grok {
    match => { "message" => "%{TIMESTAMP_ISO8601:timestamp}\s+%{LOGLEVEL:
        severity}\s+\[%{DATA:service},%{DATA:trace},%{DATA:span},%{DATA:
        exportable}\]\s+%{DATA:pid}---\s+\[%{DATA:thread}\]\s+%{DATA:class}\s+:\
        s+%{GREEDYDATA:rest}" }
  }
}
output {
  elasticsearch {
    hosts => "elasticsearch:9200"  # 改成你的Elasticsearch地址
  }
}
```

测试

1. 启动 ELK（其中，使用本节编写的 logstash.conf 启动 Logstash）。
2. 将项目 microservice-simple-provider-user-trace-elk 打包成 jar 包，并将其放在/opt/目录下。
3. 多次访问*http://localhost:8000/1*，产生一些日志。
4. 访问*http://localhost:5601*，可以看到 Kibana 的首页，如图 10-2 所示。

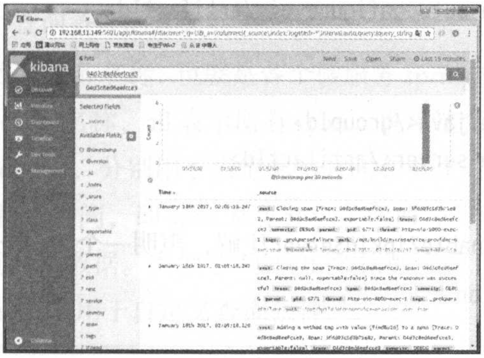


图 10-2 Kibana 首页

由图 10-2 可知，已经可以正常使用 ELK 查询、分析跟踪日志。

再次强调，必须将 spring.application.name 属性设置在 bootstrap.yml 中。

10.5 Spring Cloud Sleuth 与 Zipkin 配合使用

本节会将 Sleuth 与 Zipkin 配合使用。先来了解一下什么是 Zipkin。

10.5.1 Zipkin 简介

Zipkin 是 Twitter 开源的分布式跟踪系统，基于 Dapper 的论文设计而来。它的主要功能是收集系统的时序数据，从而追踪微服务架构的系统延时等问题。Zipkin 还提供了一个非常友好的界面，来帮助分析追踪数据。

> Zipkin 官方网站：*http://zipkin.io/*。

10.5.2 编写 Zipkin Server

本节将编写一个 Zipkin Server。

1. 创建一个 ArtifactId 是`microservice-trace-zipkin-server` 的 Maven 工程，并为项目添加以下依赖。

```
<dependency>
  <groupId>io.zipkin.java</groupId>
  <artifactId>zipkin-autoconfigure-ui</artifactId>
</dependency>
<dependency>
  <groupId>io.zipkin.java</groupId>
  <artifactId>zipkin-server</artifactId>
</dependency>
```

2. 编写启动类，使用`@EnableZipkinServer`注解，声明一个 Zipkin Server。

```
@SpringBootApplication
@EnableZipkinServer
public class ZipkinServerApplication {
  public static void main(String[] args) {
    SpringApplication.run(ZipkinServerApplication.class, args);
  }
}
```

3. 编写配置文件，在 application.yml 中添加如下内容。

```
server:
  port: 9411
```

测试

1. 启动项目 microservice-trace-zipkin-server。
2. 访问*http://localhost:9411/*，可以看到如图 10-3 的界面。

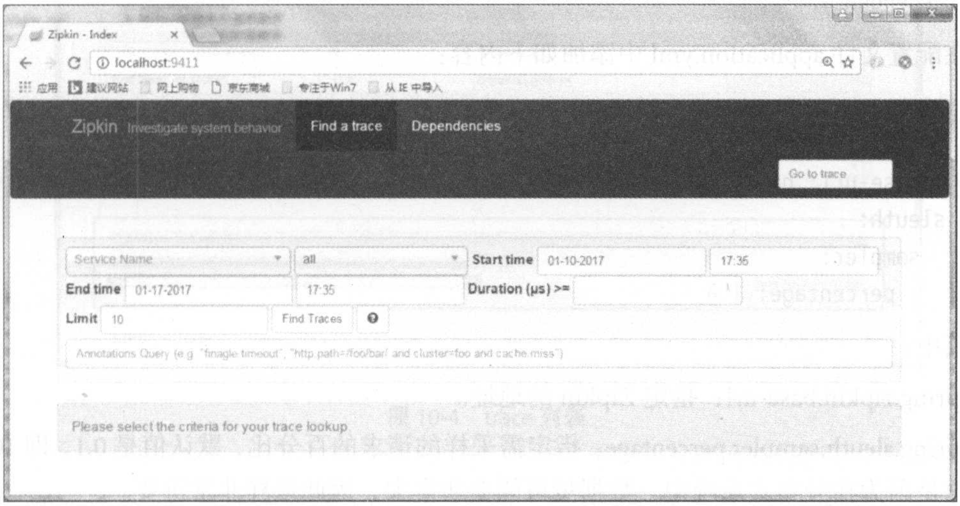


图 10-3 Zipkin Server 首页

简单讲解图中各个查询条件的含义：

- Service Name 表示服务名称，也就是各个微服务 spring.application.name 的值。
- 第二列表示 span 的名称，all 表示所有 span，也可选择指定 span。
- Start time、End time，分别用于指定起始时间和截止时间。
- Duration 表示持续时间，即 span 从创建到关闭所经历的时间。
- Limit 表示查询几条数据。类似于 MySQL 数据库中的 limit 关键词。
- Annotations Query，用于自定义查询条件。

10.5.3 微服务整合 Zipkin

本节来为微服务整合 Zipkin，以项目 microservice-simple-provider-user-trace 为例。

1. 复制项目`microservice-simple-provider-user-trace`，将 ArtifactId 修改为`microservice-simple-provider-user-trace-zipkin`。

2. 为项目添加以下依赖：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-sleuth-zipkin</artifactId>
</dependency>
```

3. 在配置文件 application.yml 中添加如下内容：

```
spring:
  zipkin:
    base-url: http://localhost:9411
  sleuth:
    sampler:
      percentage: 1.0
```

其中：

spring.zipkin.base-url：指定 Zipkin 的地址。

spring.sleuth.sampler.percentage：指定需采样的请求的百分比，默认值是 0.1，即 10%。这是因为在分布式系统中，数据量可能会非常大，因此采样非常重要。

这样就为项目整合了 Zipkin。同理，为项目 microservice-simple-consumer-movie-trace 整合 Zipkin，记为 microservice-simple-consumer-movie-trace-zipkin。

测试

1. 启动项目 microservice-trace-zipkin-server。
2. 启动项目 microservice-simple-provider-user-trace-zipkin。
3. 启动项目 microservice-simple-consumer-movie-trace-zipkin。
4. 访问*http://localhost:8010/user/1*，可正常获得结果。
5. 访问 Zipkin Server 首页*http://localhost:9411/*，填入起始时间、结束时间等筛选条件后，单击 Find a trace 按钮，可看到 trace 列表，如图 10-4 所示。

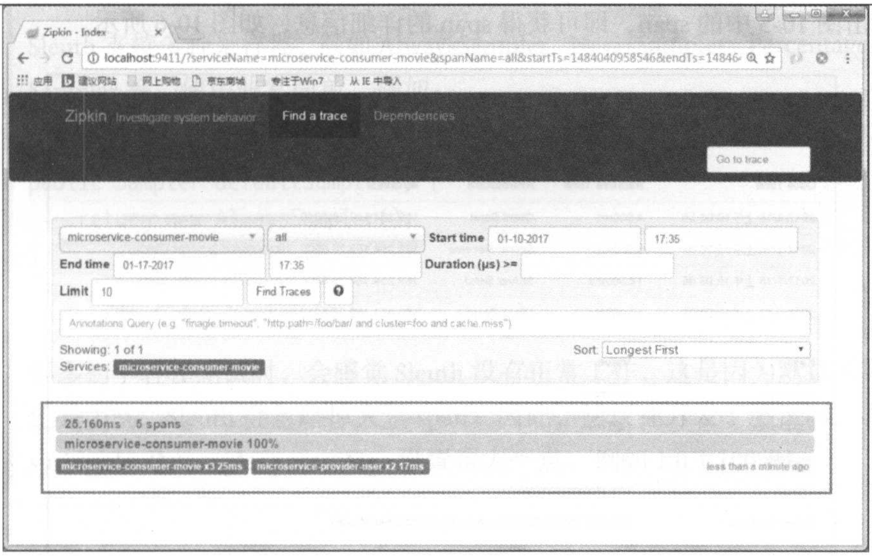


图 10-4 trace 列表

6. 单击该 trace，可看到如图 10-5 所示界面，该图详细展示了请求的细节。

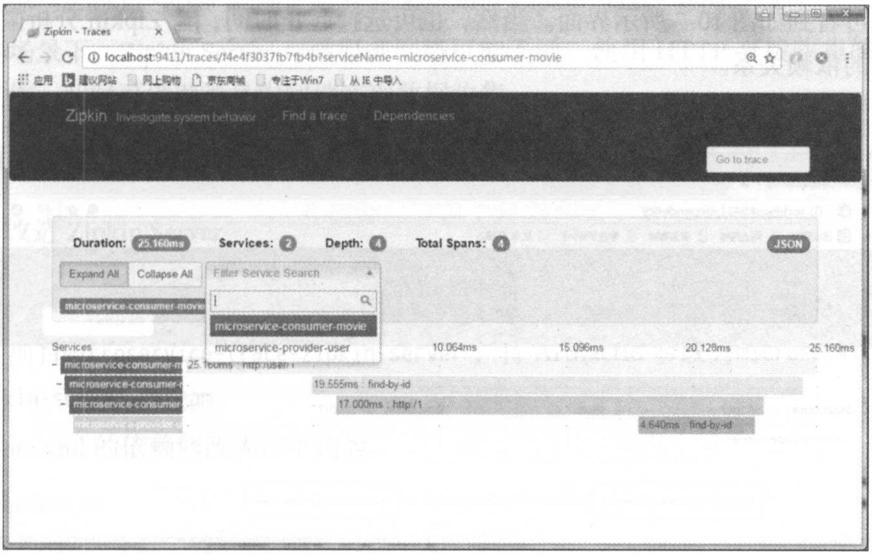


图 10-5 trace 详情

7. 单击图 10-5 中的 span，即可获得 span 的详细信息，如图 10-6 所示。

图 10-6　span 详情

8. Zipkin 还有助于分析微服务间的依赖关系。单击导航栏上的 Dependencies 按钮，即可看到如图 10-7 所示界面。当然，也可选择起止时间，让 Zipkin 分析微服务间的依赖关系。

图 10-7　微服务依赖关系

Sleuth 支持多种采样器，例如 AlwaysSampler、NeverSampler、PercentageBasedSampler 等。使用非常简单，例如：

```
@Bean
public Sampler defaultSampler() {
    return new AlwaysSampler();
}
```

很多初学者在测试时，会感觉 Sleuth 没有正常工作，这是因为默认采样百分比是 10%，Sleuth 会忽略掉大量 span。因此，建议在开发、测试时，将属性 spring.sleuth.sampler.percentage 设置得大一点，例如 1.0（100%）。

10.5.4 使用消息中间件收集数据

前文是使用 HTTP 直接收集跟踪数据的，本节来讨论如何使用消息中间件收集追踪数据。相比 HTTP 的方式来说，使用消息中间件有以下优点：

- 微服务与 Zipkin Server 解耦，微服务无须知道 Zipkin Server 的网络地址。
- 一些场景下，Zipkin Server 与微服务网络可能不通，使用 HTTP 直接收集的方式无法工作，此时可借助消息中间件实现数据收集。

笔者以 RabbitMQ 作为消息中间件进行演示，RabbitMQ 的安装详见 7.5.3.1 节。

10.5.4.1 改造 Zipkin Server

先来改造 Zipkin Server。

1. 复制项目`microservice-trace-zipkin-server`，将 ArtifactId 修改为`microservice-trace-zipkin-server-stream`。
2. 将 pom.xml 的依赖修改为以下内容。

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-sleuth-zipkin-stream</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-sleuth</artifactId>
```

```
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
</dependency>
<dependency>
  <groupId>io.zipkin.java</groupId>
  <artifactId>zipkin-autoconfigure-ui</artifactId>
</dependency>
```

3. 修改启动类，将注解 @EnableZipkinServer 修改为 @EnableZipkinStreamServer。
4. 将配置文件 application.yml 修改为如下内容。

```
server:
  port: 9411
spring:
  rabbitmq:
    host: localhost
    port: 5672
    username: guest
    password: guest
```

这样，Zipkin Server 就改造完成了。

10.5.4.2 改造微服务

改造完 Zipkin Server 后，接下来改造前文编写的微服务。

1. 复制项目microservice-simple-provider-user-trace-zipkin，将 ArtifactId 修改为 microservice-simple-provider-user-trace-zipkin-stream。
2. 修改 pom.xml，添加以下依赖。

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-sleuth-stream</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
<dependency>
```

```
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
</dependency>
```

3. 修改配置文件 application.yml，删除其中的：

```
spring:
  zipkin:
    base-url: http://localhost:9411
```

添加如下内容：

```
spring:
  rabbitmq:
    host: localhost
    port: 5672
    username: guest
    password: guest
```

这样，微服务就改造完成了。同理，改造电影微服务，详见本书配套代码中的项目 microservice-simple-consumer-movie-trace-zipkin-stream。

依次启动 microservice-trace-zipkin-server-stream、microservice-simple-provider-user-trace-zipkin-stream、microservice-simple-consumer-movie-trace-zipkin-stream 后，按照前文的步骤测试，会发现依然可以正常跟踪微服务的调用。

10.5.5 存储跟踪数据

前文的示例中，Zipkin Server 是将数据存储在内存中的。这种方式一般不适用于生产环境，因为一旦 Zipkin Server 重启或发生崩溃，就会导致历史数据的丢失。

Zipkin Server 支持多种后端存储，例如 MySQL、Elasticsearch、Cassandra 等。本节将讨论如何将数据存储在 Elasticsearch 5.1.2 中。

用项目 microservice-trace-zipkin-server-stream 进行改造，让其使用 RabbitMQ 收集跟踪数据并使用 Elasticsearch 5.1.2 作为后端存储。

1. 复制项目`microservice-trace-zipkin-server-stream`，将 ArtifactId 修改为`microservice-trace-zipkin-server-stream-elasticsearch`。
2. 将 pom.xml 的依赖修改为以下内容。

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
```

```
  <artifactId>spring-cloud-sleuth-zipkin-stream</artifactId>
</dependency>
<dependency>
  <groupId>io.zipkin.java</groupId>
  <artifactId>zipkin-autoconfigure-ui</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
</dependency>
<dependency>
  <groupId>io.zipkin.java</groupId>
  <artifactId>zipkin-autoconfigure-storage-elasticsearch-http</artifactId>
  <version>1.16.2</version>
</dependency>
```

3. 修改配置文件 application.yml，添加如下内容。

```
zipkin:
  storage:
    type: elasticsearch
    elasticsearch:
      cluster: elasticsearch
      hosts: http://localhost:9200
      index: zipkin
      index-shards: 5
      index-replicas: 1
```

这样，代码就改造完成了。

测试

1. 启动 Elasticsearch 5.1.2。
2. 启动项目 microservice-trace-zipkin-server-stream-elasticsearch。
3. 启动项目 microservice-simple-provider-user-trace-zipkin-stream。
4. 启动项目 microservice-simple-consumer-movie-trace-zipkin-stream。
5. 按照前文讲解的方式测试，可获得预期结果。
6. 访问 Zipkin 首页，可正常显示跟踪数据。
7. 访问*http://localhost:9200/_search*，可看到类似如下的结果。

```
{
  "took": 16,
  "timed_out": false,
  "_shards": {
      "total": 5,
      "successful": 5,
      "failed": 0
  },
  "hits": {
      "total": 12,
      "max_score": 1,
      "hits": [
          {
              "_index": "zipkin-2017-01-17",
              ...
}
```


说明能够正常将数据存储在 Elasticsearch 5.1.2 中。

8. 重启 Zipkin Server，在 Zipkin Server 首页输入条件查询，仍可查询到历史数据，说明可以正常从 Elasticsearch 5.1.2 中读取数据。

11 Spring Cloud 常见问题与总结

在使用 Spring Cloud 的过程中，可能会遇到一些问题。事实上，不少问题已在前面的章节中以 WARNING 的形式标出。

本章来对 Spring Cloud 的常见问题做一些总结。

11.1 Eureka 常见问题

本节将总结 Eureka 使用中常会遇到的一些问题。

11.1.1 Eureka 注册服务慢

默认情况下，服务注册到 Eureka Server 的过程较慢。在开发或测试时，常常希望能够加速这一过程，从而提升工作效率。

Spring Cloud 官方文档详细描述了该问题的原因并提供了解决方案：

> Why is it so Slow to Register a Service?
>
> Being an instance also involves a periodic heartbeat to the registry (via the client's serviceUrl) with default duration 30 seconds. A service is not available for discovery by clients until the instance, the server and the client all have the same metadata in

their local cache (so it could take 3 heartbeats). You can change the period using eureka.instance.leaseRenewalIntervalInSeconds and this will speed up the process of getting clients connected to other services. In production it's probably better to stick with the default because there are some computations internally in the server that make assumptions about the lease renewal period.

简单翻译一下：服务的注册涉及到周期性心跳，默认 30 秒一次（通过客户端配置的 serviceUrl）。只有当实例、服务器端和客户端的本地缓存中的元数据都相同时，服务才能被其他客户端发现（所以可能需要 3 次心跳）。可以使用参数 eureka.instance.leaseRenewalIntervalInSeconds 修改时间间隔，从而加快客户端连接到其他服务的过程。在生产环境中最好坚持使用默认值，因为在服务器内部有一些计算，它们会对续约做出假设。

综上，要想解决服务注册慢的问题，只须将 `eureka.instance.leaseRenewalIntervalInSeconds` 设成一个更小的值。该配置用于设置 Eureka Client 向 Eureka Server 发送心跳的时间间隔，默认是 30，单位是秒。在生产环境中，建议坚持使用默认值。

原文来自：*http://cloud.spring.io/spring-cloud-static/Camden.SR1/#_why_is_it_so_slow_to_register_a_service*。

11.1.2 已停止的微服务节点注销慢或不注销

在开发环境下，常常希望 Eureka Server 能迅速有效地注销已停止的微服务实例。然而，由于 Eureka Server 清理无效节点周期长（默认 90 秒），以及自我保护模式等原因，可能会遇到微服务注销慢甚至不注销的问题。解决方案如下：

- Eureka Server 端：

  配置关闭自我保护，并按需配置 Eureka Server 清理无效节点的时间间隔。

  ```
  eureka.server.enable-self-preservation
  # 设为false，关闭自我保护，从而保证会注销微服务
  eureka.server.eviction-interval-timer-in-ms
  # 清理间隔（单位毫秒，默认是60*1000）
  ```

- Eureka Client 端：

  配置开启健康检查，并按需配置续约更新时间和到期时间。

```
eureka.client.healthcheck.enabled
# 设为true，开启健康检查（需要spring-boot-starter-actuator依赖）
eureka.instance.lease-renewal-interval-in-seconds
# 续约更新时间间隔（默认30秒）
eureka.instance.lease-expiration-duration-in-seconds
# 续约到期时间（默认90秒）
```

值得注意的是，这些配置仅建议在开发或测试时使用，生产环境建议坚持使用默认值。

示例

- Eureka Server 配置：

```yaml
eureka:
  server:
    enable-self-preservation: false
    eviction-interval-timer-in-ms: 4000
```

- Eureka Client 配置：

```yaml
eureka:
  client:
    healthcheck:
      enabled: true
  instance:
    lease-expiration-duration-in-seconds: 30
    lease-renewal-interval-in-seconds: 10
```

> 修改 Eureka 的续约频率可能会打破 Eureka 的自我保护特性，详见：*https://github.com/spring-cloud/spring-cloud-netflix/issues/373*。这意味着在生产环境中，如果想要使用 Eureka 的自我保护特性，应坚持使用默认配置。

11.1.3 如何自定义微服务的 Instance ID

本节来探讨如何自定义微服务的 Intance ID。Instance ID 用于唯一标识注册到 Eureka Server 上的微服务实例。

在 Eureka Server 的首页可以直观地看到各个微服务的 Instance ID。例如，图 11-1 中的`itmuch:microservice-provider-user:8000`就是 Intance ID。

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
MICROSERVICE-PROVIDER-USER	n/a (1)	(1)	UP (1) - itmuch:microservice-provider-user:8000

图 11-1 Eureka Server 上的微服务列表

在 Spring Cloud 中，服务的 Instance ID 的默认值是`${spring.cloud.client.hostname}:${spring.application.name}:${spring.application.instance_id:${server.port}}`。如果想要自定义这部分的内容，只须在微服务中配置`eureka.instane.instance-id`属性即可，例如：

```
spring:
  application:
    name: microservice-provider-user
eureka:
  instance:
    # 将Instance ID设置成IP:端口的形式
    instance-id: ${spring.cloud.client.ipAddress}:${server.port}
```

这样，就可将微服务`microservice-provider-user`的 Instance ID 设为 IP: 端口的形式。这样设置后，效果如图 11-2 所示。

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
MICROSERVICE-PROVIDER-USER	n/a (1)	(1)	UP (1) - 192.168.0.59:8000

图 11-2 Eureka Server 上的微服务列表

Spring Cloud 初始化 Instance ID 的相关代码：

- org.springframework.cloud.netflix.eureka.EurekaClientAutoConfiguration
- org.springframework.cloud.commons.util.IdUtils.getDefaultInstanceId(PropertyResolver)
- org.springframework.cloud.netflix.eureka.EurekaInstanceConfigBean.getInstanceId()

11.1.4 Eureka 的 UNKNOWN 问题总结与解决

注册信息 UNKNOWN，是新手常会遇到的问题。如图 11-3，有两种 UNKNOWN 的情况，一种是应用名称 UNKNOWN，另一种是应用状态 UNKNOWN。下面分别讨论这两种情况。

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
MICROSERVICE-CONFIG-SERVER-EUREKA	n/a (1)	(1)	UP (1) - itmuch:microservice-config-server-eureka:8080
MICROSERVICE-FOO	n/a (1)	(1)	UNKNOWN (1) - itmuch:microservice-foo:8081
UNKNOWN	n/a (1)	(1)	UP (1) - itmuch:8000

图 11-3　Eureka Server 上的微服务列表

应用名称 UNKNOWN

应用名称 UNKNOWN 显然不合适，首先是微服务的名称不够语义化，无法直观看出这是哪个微服务；更重要的是，我们常常使用应用名称消费对应微服务的接口。

一般来说，有两种情况会导致该问题的发生：

- 未配置`spring.application.name`或者`eureka.instance.appname`属性。如果这两个属性均不配置，就会导致应用名称 UNKNOWN 的问题。
- 某些版本的 SpringFox 会导致该问题，例如 SpringFox 2.6.0。建议使用 SpringFox 2.6.1 或更新版本。

微服务实例状态 UNKNOWN

微服务实例的状态 UNKNOWN 同样很麻烦。一般来讲，只会请求状态是 UP 的微服务。该问题一般由健康检查导致。

`eureka.client.healthcheck.enabled=true`必须设置在 application.yml 中，而不能设置在 bootstrap.yml 中，否则一些场景下会导致应用状态 UNKNOWN 的问题。

> - SpringFox 是一款基于 Spring 和 Swagger 的开源的 API 文档框架，前身是 swagger-springmvc。官网网站：*http://springfox.io/*。
> - Swagger 是一款非常流行的 API 文档框架，它可帮助我们设计、构建、测试 RESTful 接口，也可生成 RESTful 接口文档。官方网站：*http://swagger.io/*。

11.2 Hystrix/Feign 整合 Hystrix 后首次请求失败

某些场景下，Feign 或 Ribbon 整合 Hystrix 后，会出现首次调用失败的问题。本节将对该问题作一些总结。

11.2.1 原因分析

Hystrix 默认的超时时间是 1 秒，如果在 1 秒内得不到响应，就会进入 fallback 逻辑。由于 Spring 的懒加载机制，首次请求往往会比较慢，因此在某些机器（特别是配置低的机器）上，首次请求需要的时间可能就会大于 1 秒。

了解原因后，来总结如何解决该问题。

11.2.2 解决方案

有很多方式解决该问题，以下列举几种比较简单的方案。

- 方法一，延长 Hystrix 的超时时间，示例：

  ```
  hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds: 5000
  ```

  该配置让 Hystrix 的超时时间改为 5 秒。

- 方法二，禁用 Hystrix 的超时，示例：

  ```
  hystrix.command.default.execution.timeout.enabled: false
  ```

- 方法三，对于 Feign，还可为 Feign 禁用 Hystrix，示例：

  ```
  feign.hystrix.enabled: false
  ```

  这样即可为 Feign 全局禁用 Hystrix 支持。该方式比较极端，一般不建议使用。

11.3 Turbine 聚合的数据不完整

在某些版本的 Spring Cloud（例如 Brixton SR5）中，Turbine 会发生该问题。该问题的直观体现是：使用 Turbine 聚合了多个微服务，但在 Hystrix Dashboard 上只能看到部分微服务的监控数据。

例如 Turbine 配置如下：

```
turbine:
  appConfig: microservice-consumer-movie,microservice-consumer-movie-feign-hystrix-
      fallback-stream
  clusterNameExpression: "'default'"
```

Turbine 理应聚合 microservice-consumer-movie 和 microservice-consumer-movie-feign-hystrix-fallback-stream 这两个微服务的监控数据，然而打开 Hystrix Dashboard 时，会发现 Dashboard 上只显示部分微服务的监控数据，如图 11-4 所示。

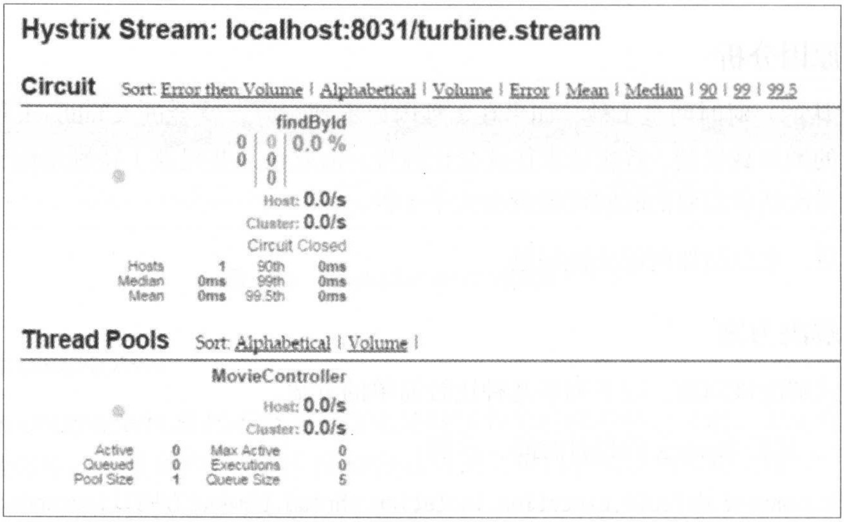


图 11-4 Hystrix Dashboard 监控页面

这显然不正常，那么如何解决这个问题呢？

解决方案

当 Turbine 聚合的微服务部署在同一台主机上时，就会出现该问题。

解决方案如下：

- 方法一：为各个微服务配置不同的 hostname，并将 preferIpAddress 设为 false 或者不设置。

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://discovery:8761/eureka/
  instance:
    hostname: ribbon # 配置hostname
```

- 方法二：设置`turbine.combine-host-port = true`。

```
turbine:
  appConfig: microservice-consumer-movie,microservice-consumer-movie-feign-
    hystrix-fallback-stream
  clusterNameExpression: "'default'"
  combine-host-port: true
```

- 方法三：升级 Spring Cloud 到 Camden 或更新版本。当然，也可单独升级 Spring Cloud Netflix 到 1.2.0 或更新版本（一般不建议单独升级 Spring Cloud Netflix，因为可能会跟 Spring Cloud 其他组件冲突）。

  这是因为老版本中的`turbine.combine-host-port`默认值是 false。Spring Cloud 已经意识到该问题，所以在新的版本中将该属性的默认值修改为 true。该解决方案和方法二本质上是一致的。

> - 相关代码：
>   `org.springframework.cloud.netflix.turbine.TurbineProperties.combineHostPort`
>   `org.springframework.cloud.netflix.turbine.CommonsInstanceDiscovery.getInstance(String, String, String, Boolean)`
> - 相关 Issue：*https://github.com/spring-cloud/spring-cloud-netflix/issues/1087*。

> 本书所使用的 Spring Cloud Camden SR4 不存在该问题。

11.4 Spring Cloud 各组件配置属性

经过本书讲解，相信大家已经发现，Spring Cloud 中的大部分问题都可使用配置属性来解决。本节会将相关组件的配置的地址罗列出来，方便读者查阅与检索。

11.4.1 Spring Cloud 的配置

Spring Cloud 的所有组件配置都在其官方文档的附录，地址如下：

http://cloud.spring.io/spring-cloud-static/Camden.SR4/#_appendix_compendium_of_configuration_properties

11.4.2 原生配置

Spring Cloud 整合了很多类库，例如 Eureka、Ribbon、Feign 等。这些组件自身也有一些配置属性，如下：

- Eureka 的配置：*https://github.com/Netflix/eureka/wiki/Configuring-Eureka*。
- Ribbon 的配置：*https://github.com/Netflix/ribbon/wiki/Programmers-Guide*。
- Hystrix 的配置：*https://github.com/Netflix/Hystrix/wiki/Configuration*。
- Turbine 的配置：*https://github.com/Netflix/Turbine/wiki/Configuration-(1.x)*。

11.5 Spring Cloud 定位问题思路总结

本节对如何定位 Spring Cloud 问题做一些总结。

根据笔者观察，Spring Cloud 进入 Camden 时代后，已经比较稳定。一般来说，问题都不是 Spring Cloud 本身的 Bug 导致。因此，读者排查问题的思路不妨按照以下步骤展开。

1. 排查配置问题

   排查配置有无问题，举几个简单的例子。

   - **YAML 缩进是否正确**

     曾经有朋友发现 Spring Cloud 应用程序无法正常启动，或配置无法正常加载。经笔者协助，发现仅仅是 YAML 配置文件缩进不正确。

     类似问题应在编码的过程中严格规避。

   - **配置属性是否正确**

     配置的属性写错，也是一个非常常见的问题。尽管该问题很低级，但从笔者的观察来看，不少初学者都会遇到这类问题。

     很多场景下，这类问题可借助 IDE 的提示功能来排查——当 IDE 不自动提示或给出警告时，应格外注意。

   - **配置属性的位置是否正确**

     配置属性位置不正确可能会导致应用的不正常。举几个常见的例子：

     - 应当配置在 Eureka Client 项目上的属性，配置在了 Eureka Server 项目上。
     - 应当写在 bootstrap.yml 中的属性，写在了 application.yml 中，例如：

```
spring:
  cloud:
    config:
```

```
uri: http://localhost:8080/
```

该属性应当存放在 bootstrap.yml 中。

- 应当写在 application.yml 的属性，写在了 bootstrap.yml 中，例如：

```
eureka.client.healthcheck.enabled=true
```

2. 排查环境问题

如确认配置无误，即可考虑运行环境是否存在问题。举几个例子：

- **环境变量**

  例如 Java 环境变量、Maven 环境变量以及 Docker 容器环境变量等。当应用无法正常工作时，应该确保环境变量配置正确。

- **依赖下载是否完整**

  曾经有朋友遇到应用无法正常启动的问题，最终发现仅仅是依赖没有下载完整所致。因此，建议在启动应用前，使用以下命令打包，从而确认依赖的完整性。

  ```
  mvn clean package
  ```

- **网络问题**

  微服务之间通过网络保持通信，因此，网络常常是排查问题的关键。当问题发生时，可优先排查网络问题。

3. 排查代码问题

如经过以上步骤，依然没有定位到 Spring Cloud 的问题，那么可能是编写的代码出了问题。很多时候，常常因为少了某个注解，或是依赖缺失，而导致了各种异常。许多场景下，设置合理的日志级别，会对问题的定位有奇效。

4. 排查 Spring Cloud 自身的问题

如果确定不是自身代码问题，就可 Debug 一下 Spring Cloud 的代码了。同时，可在 GitHub 等平台给 Spring Cloud 项目组提交 Issue，然后参考官方回复，尝试规避相应问题。如问题无法规避，就需要 Spring Cloud 进行扩展，或者修复 Spring Cloud 的 Bug，从而满足需求。此时，请不要忘记在 Spring Cloud 的 Github 上 Pull Request，协助官方改进 Spring Cloud，让 Spring Cloud 更加完善、稳定。

> 可供参考的资源：
>
> - 各项目自身的 GitHub，例如 Eureka 的 GitHub：*https://github.com/Netflix/eureka*。

- Spring Cloud 对应项目的 GitHub，例如 Eureka 项目在 Spring Cloud Netflix 中：*https://github.com/spring-cloud/spring-cloud-netflix*。
- Spring Cloud 的 StackOverflow：*http://stackoverflow.com/questions/tagged/spring-cloud*。
- Spring Cloud 的 Gitter：*https://gitter.im/spring-cloud/spring-cloud*。
- Spring Cloud 中国社区：*http://springcloud.cn*。

在这些地方，均有官方人员参与，可帮助我们迅速解决问题。

12 Docker 入门

12.1 Docker 简介

Docker 是一个开源的容器引擎，它有助于更快地交付应用。Docker 可将应用程序和基础设施层隔离，并且能将基础设施当作程序一样进行管理。使用 Docker，可更快地打包、测试以及部署应用程序，并可以缩短从编写到部署运行代码的周期。

- Docker 的官方网站：*https://www.docker.com/*。
- Docker 的 GitHub：*https://github.com/docker/docker*。

12.2 Docker 的架构

看一下来自 Docker 官方文档的架构图，如图 12-1 所示。

接下来讲解一下图中包含的组件。

- Docker daemon（Docker 守护进程）

  Docker daemon 是一个运行在宿主机（DOCKER_HOST）的后台进程。可通过 Docker 客户端与之通信。

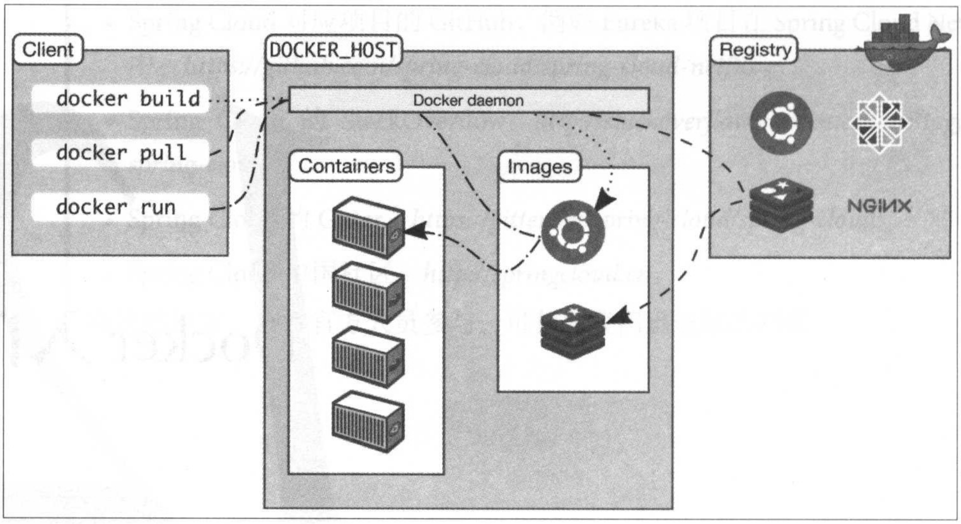


图 12-1 Docker 架构图

- Client（Docker 客户端）

  Docker 客户端是 Docker 的用户界面，它可以接受用户命令和配置标识，并与 Docker daemon 通信。图中，docker build 等都是 Docker 的相关命令。

- Images（Docker 镜像）

  Docker 镜像是一个只读模板，它包含创建 Docker 容器的说明。它和系统安装光盘有点像——使用系统安装光盘可以安装系统，同理，使用 Docker 镜像可以运行 Docker 镜像中的程序。

- Container（容器）

  容器是镜像的可运行实例。镜像和容器的关系有点类似于面向对象中，类和对象的关系。可通过 Docker API 或者 CLI 命令来启停、移动、删除容器。

- Registry

  Docker Registry 是一个集中存储与分发镜像的服务。构建完 Docker 镜像后，就可在当前宿主机上运行。但如果想要在其他机器上运行这个镜像，就需要手动复制。此时可借助 Docker Registry 来避免镜像的手动复制。

  一个 Docker Registry 可包含多个 Docker 仓库，每个仓库可包含多个镜像标签，每个标签对应一个 Docker 镜像。这跟 Maven 的仓库有点类似，如果把 Docker Registry 比作 Maven 仓库的话，那么 Docker 仓库就可理解为某 jar 包的路径，而镜像标签则可理解为 jar 包的版本号。

Docker Registry 可分为公有 Docker Registry 和私有 Docker Registry。最常用的 Docker Registry 莫过于官方的 Docker Hub，这也是默认的 Docker Registry。Docker Hub 上存放着大量优秀的镜像，可使用 Docker 命令下载并使用。

12.3 安装 Docker

Docker 官方建议将 Docker 运行在 Linux 操作系统上。当然，Docker 也可运行在其他的平台，例如 Windows、Mac OS 等。

本节将演示如何在 CentOS 上安装 Docker，其他操作系统上的安装可参考官方文档：*https://docs.docker.com/engine/installation/*。

12.3.1 系统要求

- Docker 运行在 CentOS 7.X 之上。
- Docker 需要安装在 64 位平台。

12.3.2 移除非官方软件包

Red Hat 操作系统包含了一个旧版本的 Docker 软件包，该旧版本软件包的名称是“docker”（新版是“docker-engine”）。因此，如已安装该软件包，那么需要执行以下命令移除。

```
sudo yum -y remove docker
```

执行该命令只会移除旧版本的 Docker，/var/lib/docker 目录中的内容不会被删除，因此，旧版本 Docker 所创建的镜像、容器、卷等都会保留下来。

12.3.3 设置 Yum 源

Docker 有多种安装方式，例如 Yum 安装、RPM 包安装、Shell 安装等。本节以 Yum 为例进行讲解。

1. 安装 yum-utils，这样就能使用 yum-config-manager 工具设置 Yum 源。

```
sudo yum install -y yum-utils
```

2. 执行以下命令，添加 Docker 的 Yum 源。

```
sudo yum-config-manager \
   --add-repo \
   https://docs.docker.com/engine/installation/linux/repo_files/centos/docker.
repo
```

3. **[可选]** 启用测试仓库。测试仓库包含在 docker.repo 文件中，但默认情况下是禁用的。如需启用测试仓库，可使用以下命令：

```
sudo yum-config-manager --enable docker-testing
```

想要禁用测试仓库，可执行以下命令：

```
sudo yum-config-manager --disable docker-testing
```

12.3.4 安装 Dokcer

1. 更新 Yum 包的索引。

```
sudo yum makecache fast
```

2. 安装最新版本的 Docker。

```
sudo yum -y install docker-engine
```

这样，经过一段时间的等待后，Docker 就安装完成了。

3. 在生产系统中，可能需要安装指定版本的 Docker，而并不总是安装最新版本。执行以下命令，即可列出可用的 Docker 版本。

```
yum list docker-engine.x86_64  --showduplicates |sort -r
```

其中，sort -r 命令表示对结果由高到低排序。执行后，可看到类似于如下的表格：

```
docker-engine.x86_64            1.13.0-1.el7.centos           docker-main
docker-engine.x86_64            1.12.6-1.el7.centos           docker-main
docker-engine.x86_64            1.12.5-1.el7.centos           docker-main
...
```

该表格有三列，第一列是软件包名称，第二列是版本字符串，第三列是仓库名称，表示软件包存储的位置，例如 docker-main、docker-testing 等。列出 Docker 版本后，可使用以下命令安装指定版本的 Docker。

```
sudo yum -y install docker-engine-<VERSION_STRING>
```

例如：

```
sudo yum -y install docker-engine-1.13.0
```

4. 启动 Dokcer。

```
sudo systemctl start docker
```

5. 执行以下命令，验证安装是否正确。

```
sudo docker run hello-world
```

如看到类似于如下的结果，则说明安装正确。

```
Unable to find image 'hello-world:latest' locally
...
Hello from Docker!
This message shows that your installation appears to be working correctly.
...
```

6. 查看 Docker 版本。

```
docker version
```

可看到类似于如下的结果：

```
Client:
 Version:      1.13.0
 API version:  1.25
 Go version:   go1.7.3
 Git commit:   49bf474
 Built:        Tue Jan 17 09:55:28 2017
 OS/Arch:      linux/amd64
Server:
 Version:      1.13.0
 API version:  1.25 (minimum version 1.12)
 Go version:   go1.7.3
 Git commit:   49bf474
 Built:        Tue Jan 17 09:55:28 2017
 OS/Arch:      linux/amd64
 Experimental: false
```

由结果可知当前 Docker 版本、API 版本、Go 语言版本等信息。

12.3.5 卸载 Docker

1. 卸载 Docker 软件包。

```
sudo yum -y remove docker-engine
```

2. 如需删除镜像、容器、卷以及自定义的配置文件，可执行以下命令：

```
sudo rm -rf /var/lib/docker
```

12.4 配置镜像加速器

国内访问 Docker Hub 的速度很不稳定，有时甚至出现连接不上的情况。本节来为 Docker 配置镜像加速器，从而解决这个问题。目前国内很多云服务商都提供了镜像加速的服务。

常用的镜像加速器有：阿里云加速器、DaoCloud 加速器等。各厂商镜像加速器的使用方式大致类似，本节以阿里云加速器为例进行讲解。

1. 注册阿里云账号后，即可在阿里云控制台（*https://cr.console.aliyun.com/#/accelerator*）看到如图 12-2 的页面。

图 12-2 阿里云管理控制台

2. 按照图 12-2 的说明，即可配置镜像加速器。

12.5 Docker 常用命令

Docker 有很多命令，这些命令有助于控制 Docker 的行为。本节将详细探讨 Docker 常用命令。

12.5.1 Docker 镜像常用命令

首先来讨论 Docker 镜像的常用命令。

- 搜索镜像

  可使用docker search 命令搜索存放在 Docker Hub 中的镜像。例如：

```
docker search java
```

  执行该命令后，Docker 就会在 Docker Hub 中搜索含有 java 这个关键词的镜像仓库。执行该命令后，可看到类似于如下的表格：

```
NAME                  DESCRIPTION                    STARS  OFFICIAL  AUTOMATED
java                  Java is a concurrent, ...      1281   [OK]
anapsix/alpine-java   Oracle Java 8 (and 7) ...      190              [OK]
isuper/java-oracle    This repository conta ...      48               [OK]
lwieske/java-8        Oracle Java 8 Contain ...      32               [OK]
nimmis/java-centos    This is docker images ...      23               [OK]
...
```

  该表格包含五列，含义如下。

  - NAME：镜像仓库名称。
  - DESCRIPTION：镜像仓库描述。
  - STARS：镜像仓库收藏数，表示该镜像仓库的受欢迎程度，类似于 GitHub 的 Stars。
  - OFFICAL：表示是否为官方仓库，该列标记为 [OK] 的镜像均由各软件的官方项目组创建和维护。由结果可知，java 这个镜像仓库是官方仓库，而其他的仓库都不是镜像仓库。
  - AUTOMATED：表示是否是自动构建的镜像仓库。

- 下载镜像

  使用命令docker pull 命令即可从 Docker Registry 上下载镜像，例如：

```
docker pull java
```

  执行该命令后，Docker 会从 Docker Hub 中的 java 仓库下载最新版本的 Java 镜像。若镜像下载缓慢，可配置镜像加速器，详见 12.4 一节。

  该命令还可指定想要下载的镜像标签以及 Docker Registry 地址，例如：

```
docker pull reg.itmuch.com/java:7
```

  这样就可以从指定的 Docker Registry 中下载标签为 7 的 Java 镜像。

- 列出镜像

  使用`docker images`命令即可列出已下载的镜像。

  执行该命令后，将会看到类似于如下的表格：

```
REPOSITORY    TAG       IMAGE ID        CREATED         SIZE
java          latest    861e95c114d6    4 weeks ago     643.1 MB
hello-world   latest    c54a2cc56cbb    5 months ago    1.848 kB
```

  该表格包含了 5 列，含义如下。

  - REPOSITORY：镜像所属仓库名称。
  - TAG：镜像标签。默认是 latest，表示最新。
  - IMAGE ID：镜像 ID，表示镜像唯一标识。
  - CREATED：镜像创建时间。
  - SIZE：镜像大小。

- 删除本地镜像

  使用`docker rmi`命令即可删除指定镜像。

  例 1：删除指定名称的镜像。

```
docker rmi hello-world
```

  表示删除 hello-world 这个镜像。

  例 2：删除所有镜像。

```
docker rmi -f $(docker images)
```

  -f 参数表示强制删除。

> Docker 的命令：*https://docs.docker.com/engine/reference/commandline/*。

12.5.2 Docker 容器常用命令

本节来讨论 Docker 容器的常用命令。

1. 新建并启动容器

   使用以下`docker run`命令即可新建并启动一个容器。

   该命令是最常用的命令，它有很多选项，下面将列举一些常用的选项。

- -d 选项：表示后台运行
- -P 选项：随机端口映射
- -p 选项：指定端口映射，有以下四种格式。
  - ip:hostPort:containerPort
  - ip::containerPort
  - hostPort:containerPort
  - containerPort
- --network 选项：指定网络模式，该选项有以下可选参数：
  - --network=bridge：默认选项，表示连接到默认的网桥。
  - --network=host：容器使用宿主机的网络。
  - --network=container:NAME_or_ID：告诉 Docker 让新建的容器使用已有容器的网络配置。
  - --network=none：不配置该容器的网络，用户可自定义网络配置。

示例 1：

```
docker run java /bin/echo 'Hello World'
```

这样终端会打印 Hello World 的字样，跟在本地直接执行`/bin/echo 'Hello World'`一样。

示例 2：

```
docker run -d -p 91:80 nginx
```

这样就能启动一个 Nginx 容器。在本例中，为 docker run 添加了两个参数，含义如下：

```
-d # 后台运行
-p 宿主机端口:容器端口 # 开放容器端口到宿主机端口
```

访问 http://Docker 宿主机 IP:91/，将会看到如图 12-3 的界面：

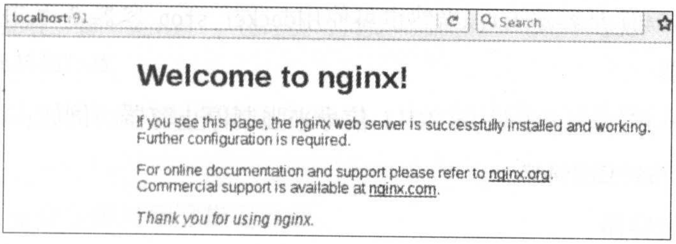


图 12-3 Nginx 首页

> 需要注意的是，使用 docker run 命令创建容器时，会先检查本地是否存在指定镜像。如果本地不存在该名称的镜像，Docker 就会自动从 Docker Hub 下载镜像并启动一个 Docker 容器。

2. 列出容器

使用docker ps 命令即可列出运行中的容器。执行该命令后，可看到类似于如下的表格。

```
CONTAINER ID  IMAGE  COMMAND                    CREATED         STATUS      PORTS
                     NAMES
784fd3b294d7  nginx  "nginx -g 'daemon off"  20 minutes ago  Up 2 seconds 443/t
cp, 0.0.0.0:91->80/tcp  backstabbing_archimedes
```

如需列出所有容器（包括已停止的容器），可使用-a 参数。

该表格包含了 7 列，含义如下。

- CONTAINER_ID：表示容器 ID。
- IMAGE：表示镜像名称。
- COMMAND：表示启动容器时运行的命令。
- CREATED：表示容器的创建时间。
- STATUS：表示容器运行的状态。Up 表示运行中，Exited 表示已停止。
- PORTS：表示容器对外的端口号。
- NAMES：表示容器名称。该名称默认由 Docker 自动生成，也可使用 docker run 命令的--name 选项自行指定。

3. 停止容器

使用docker stop 命令，即可停止容器。例如：

```
docker stop 784fd3b294d7
```

其中784fd3b294d7 是容器 ID，当然也可使用docker stop 容器名称 来停止指定容器。

4. 强制停止容器

可使用docker kill 命令发送 SIGKILL 信号来强制停止容器。例如：

```
docker kill 784fd3b294d7
```

5. 启动已停止的容器

使用docker run 命令，即可新建并启动一个容器。对于已停止的容器，可使用docker start 命令来启动。例如：

```
docker start 784fd3b294d7
```

6. 重启容器

可使用docker restart 命令来重启容器。该命令实际上是先执行了docker stop 命令，然后执行了docker start 命令。

7. 进入容器

某场景下，可能需要进入运行中的容器。

- 使用docker attach 命令进入容器。例如：

  ```
  docker attach 784fd3b294d7
  ```

  很多场景下，使用docker attach 命令并不方便。当多个窗口同时 attach 到同一个容器时，所有窗口都会同步显示。同理，如果某个窗口发生阻塞，其他窗口也无法执行操作。

- 使用nsenter 进入容器。

  nsenter 工具包含在 util-linux 2.23 或更高版本中。为了连接到容器，需要找到容器第一个进程的 PID，可通过以下命令获取：

  ```
  docker inspect --format "{{.State.Pid}}" $CONTAINER_ID
  ```

获得 PID 后，就可使用 nsenter 命令进入容器了：

```
nsenter --target "$PID" --mount --uts --ipc --net --pid
```

下面给出一个完整的例子：

```
[root@localhost ~]# docker ps
CONTAINER ID  IMAGE   COMMAND                    CREATED          STATUS        PORTS
                      NAMES
784fd3b294d7  nginx   "nginx -g 'daemon off"  55 minutes ago  Up 3 minutes  443/t
cp, 0.0.0.0:91->80/tcp  backstabbing_archimedes
[root@localhost ~]# docker inspect --format "{{.State.Pid}}" 784fd3b294d7
95492
[root@localhost ~]# nsenter --target 95492 --mount --uts --ipc --net --pid
root@784fd3b294d7:/#
```

读者也可将以上两条命令封装成一个 Shell，从而简化进入容器的过程。

8. 删除容器

使用docker rm 命令即可删除指定容器。

例 1：删除指定容器。

```
docker rm 784fd3b294d7
```

该命令只能删除已停止的容器，如需删除正在运行的容器，可使用-f 参数。

例 2：删除所有的容器。

```
docker rm -f $(docker ps -a -q)
```

- Docker 的网络：*https://docs.docker.com/engine/userguide/networking/*。
- Docker 命令：*https://docs.docker.com/engine/reference/commandline/*。

13 将微服务运行在 Docker 上

13.1　使用 Dockerfile 构建 Docker 镜像

本节将讨论如何使用 Dockerfile 构建 Docker 镜像。Dockerfile 是一个文本文件，其中包含了若干条指令，指令描述了构建镜像的细节。

先来编写一个最简单的 Dockerfile。以前文下载的 Nginx 镜像为例（详见 12.5.2 节），来编写一个 Dockerfile 修改该镜像的首页。

1. 例如：

```
FROM nginx
RUN echo '<h1>Spring Cloud与Docker微服务实战</h1>' > /usr/share/nginx/html/
index.html
```

   该 Dockerfile 非常简单，其中的 FORM、RUN 都是 Dockerfile 的指令。

   FROM 指令用于指定基础镜像，RUN 指令用于执行命令。

2. 在 Dockerfile 所在路径执行以下命令构建镜像：

```
docker build -t nginx:my .
```

   其中，命令最后的点（.）用于路径参数传递，表示当前路径。

3. 执行以下命令，即可使用该镜像启动一个 Docker 容器。

```
docker run -d -p 92:80 nginx:my
```

4. 访问 http://Docker 宿主机 IP:92/，可看到如图 13-1 的界面。

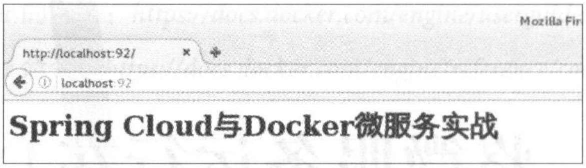


图 13-1 Nginx 首页

从本例不难看出 Dockerfile 的强大。仅仅编写了两行代码，就修改了原始镜像的行为。不仅如此，通过 Dockerfile，还可直观地看到修改镜像的具体过程。

> 除了使用 Dockerfile 构建镜像，也可手工制作 Docker 镜像，但这种方式烦琐、效率低，一般不适合生产，本书就不再赘述。

13.1.1 Dockerfile 常用指令

在前面的例子中，提到了 FORM、RUN 指令。事实上，Dockerfile 有十多个指令。本节将系统讲解这些指令，指令的一般格式为：指令名称 参数 。

1. ADD 复制文件

   ADD 指令用于复制文件，格式为：

   - ADD <src>... <dest>
   - ADD ["<src>",... "<dest>"]

   从 src 目录复制文件到容器的 dest。其中 src 可以是 Dockerfile 所在目录的相对路径，也可以是一个 URL，还可以是一个压缩包

   注意：

   - src 必须在构建的上下文内，不能使用例如：ADD ../somethine /something这样的命令，因为docker build命令首先会将上下文路径和其子目录发送到 docker daemon。
   - 如果 src 是一个 URL，同时 dest 不以斜杠结尾，dest 将会被视为文件，src 对应内容文件将会被下载到 dest。
   - 如果 src 是一个 URL，同时 dest 以斜杠结尾，dest 将被视为目录，src 对应内容将会被下载到 dest 目录。

- 如果 src 是一个目录，那么整个目录下的内容将会被复制，包括文件系统元数据。
- 如果文件是可识别的压缩包格式，则 docker 会自动解压。

示例：

```
ADD microservice-discovery-eureka-0.0.1-SNAPSHOT.jar app.jar
```

2. ARG 设置构建参数

ARG 指令用于设置构建参数，类似于 ENV。和 ARG 不同的是，ARG 设置的是构建时的环境变量，在容器运行时是不会存在这些变量的。

格式为：`ARG <name>[=<default value>]`。

示例：

```
ARG user1=someuser
```

3. CMD 容器启动命令

CMD 指令用于为执行容器提供默认值。每个 Dockerfile 只有一个 CMD 命令，如果指定了多个 CMD 命令，那么只有最后一条会被执行，如果启动容器时指定了运行的命令，则会覆盖掉 CMD 指定的命令。

支持 3 种格式：

- `CMD ["executable","param1","param2"]`（推荐使用）
- `CMD ["param1","param2"]`（为 ENTRYPOINT 指令提供预设参数）
- `CMD command param1 param2`（在 shell 中执行）

示例：

```
CMD echo "This is a test." | wc -
```

4. COPY 复制文件

复制文件，格式为：

- `COPY <src>... <dest>`
- `COPY ["<src>",... "<dest>"]`

复制本地端的 src 到容器的 dest。COPY 指令和 ADD 指令类似，COPY 不支持 URL 和压缩包。

5. ENTRYPOINT 入口点

格式为：

- `ENTRYPOINT ["executable", "param1", "param2"]`
- `ENTRYPOINT command param1 param2`

ENTRYPOINT 和 CMD 指令的目的一样，都是指定 Docker 容器启动时执行的命令，可多次设置，但只有最后一个有效。

6. ENV 设置环境变量

ENV 指令用于设置环境变量，格式为：

- `ENV <key> <value>`
- `ENV <key>=<value> ...`

示例：

```
ENV JAVA_HOME /path/to/java
```

7. EXPOSE 声明暴露的端口

EXPOSE 指令用于声明在运行时容器提供服务的端口，格式为：`EXPOSE <port> [<port>...]`。

需要注意的是，这只是一个声明，运行时并不会因为该声明就打开相应端口。该指令的作用主要是帮助镜像使用者理解该镜像服务的守护端口；其次是当运行时使用随机映射时，会自动映射 EXPOSE 的端口。示例：

```
# 声明暴露一个端口示例
EXPOSE port1
# 相应的运行容器使用的命令
docker run -p port1 image
# 也可使用-P选项启动
docker run -P image

# 声明暴露多个端口示例
EXPOSE port1 port2 port3
# 相应的运行容器使用的命令
docker run -p port1 -p port2 -p port3 image
# 也可指定需要映射到宿主机器上的端口号
docker run -p host_port1:port1 -p host_port2:port2 -p host_port3:port3 image
```

8. FROM 指定基础镜像

使用 FORM 指令指定基础镜像，FORM 指令有点像 Java 里面的 extends 关键字。需要注意的是，FROM 指令必须指定且需要写在其他指令之前。FORM 指令后的所有指令都依赖于该指令所指定的镜像。

支持 3 种格式：

- FROM <image>
- FROM <image>:<tag>
- FROM <image>@<digest>

9. LABEL 为镜像添加元数据

LABEL 指令用于为镜像添加元数据。

格式为：LABEL <key>=<value> <key>=<value> <key>=<value> ...。

使用 """ 和 "\" 转换命令行，示例：

```
LABEL "com.example.vendor"="ACME Incorporated"
LABEL com.example.label-with-value="foo"
LABEL version="1.0"
LABEL description="This text illustrates \
that label-values can span multiple lines."
```

10. MAINTAINER 指定维护者的信息

MAINTAINER 指令用于指定维护者的信息，用于为 Dockerfile 署名。

格式为：MAINTAINER <name>。

示例：

```
MAINTAINER 周立<eacdy0000@126.com>
```

11. RUN 执行命令

该指令支持两种格式：

- RUN <command>
- RUN ["executable", "param1", "param2"]

RUN <command>在 shell 终端中运行，在 Linux 中默认是/bin/sh -c，在 Windows 中是cmd /s /c，使用这种格式，就像直接在命令行中输入命令一样。RUN ["executable", "param1", "param2"]使用 exec 执行，这种方式类似于函数调用。指定其他终端可以通过该方式操作，例如：RUN ["/bin/bash", "-c", "echo hello"]，该方式必须使用双引号"而不能使用单引号'，因为该方式会被转换成一个 JSON 数组。

12. USER 设置用户

该指令用于设置启动镜像时的用户或者 UID，写在该指令后的 RUN、CMD 以及 ENTRYPOINT 指令都将使用该用户执行命令。

格式为：USER 用户名。

示例：

```
USER daemon
```

13. VOLUME 指定挂载点

该指令使容器中的一个目录具有持久化存储的功能，该目录可被容器本身使用，也可共享给其他容器。当容器中的应用有持久化数据的需求时可以在 Dockerfile 中使用该指令。格式为：`VOLUME ["/data"]`。

示例：

```
VOLUME /data
```

14. WORKDIR 指定工作目录

格式为：`WORKDIR /path/to/workdir`。

切换目录指令，类似于 cd 命令，写在该指令后的`RUN`，`CMD`以及`ENTRYPOINT`指令都将该目录作为当前目录，并执行相应的命令。

15. 其他

Dockerfile 还有一些其他的指令，例如 STOPSINGAL、HEALTHCHECK、SHELL 等。由于并不是十分常用，本书不再赘述。有兴趣的读者可前往*https://docs.docker.com/engine/reference/builder/*进行扩展阅读。

- Dockerfile 官方文档：*https://docs.docker.com/engine/reference/builder/#dockerfile-reference*。
- Dockerfile 最佳实践：*https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/#build-cache*。

13.1.2 使用 Dockerfile 构建镜像

前文详细讲解了 Dockerfile 的常用指令，本节把前文编写的 Spring Cloud 微服务构建成 Docker 镜像。

准备工作

以项目`microservice-discovery-eureka`为例，首先执行以下命令，将项目构建成 jar 包：microservice-discovery-eureka-0.0.1-SNAPSHOT.jar。

```
mvn clean package       # 使用Maven打包项目
```

使用 Dockerfile 构建 Docker 镜像

1. 在 jar 包所在目录，创建名为 Dockerfile 的文件。

```
touch Dockerfile
```

2. 在 Dockerfile 中添加以下内容。

```
# 基于哪个镜像
FROM java:8

# 将本地文件夹挂载到当前容器
VOLUME /tmp

# 复制文件到容器，也可以直接写成ADD microservice-discovery-eureka-0.0.1-
    SNAPSHOT.jar /app.jar
ADD microservice-discovery-eureka-0.0.1-SNAPSHOT.jar app.jar
RUN bash -c 'touch /app.jar'

# 声明需要暴露的端口
EXPOSE 8761

# 配置容器启动后执行的命令
ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]
```

3. 使用docker build 命令构建镜像。

```
docker build -t itmuch/microservice-discovery-eureka:0.0.1 .
# 格式：docker build -t 仓库名称/镜像名称(:标签) Dockerfile的相对位置
```

在这里，使用-t 选项指定了镜像的标签。执行该命令后，终端将会输出如下的内容。

```
Sending build context to Docker daemon 71.89 MB
Step 1/6 : FROM java:8
 ---> d23bdf5b1b1b
Step 2/6 : VOLUME /tmp
 ---> Running in da87c91b2ff7
 ---> ac325d3d36f0
Removing intermediate container da87c91b2ff7
Step 3/6 : ADD microservice-discovery-eureka-0.0.1-SNAPSHOT.jar app.jar
 ---> 11d1aeefaa05
Removing intermediate container 37eacdcd1a9a
Step 4/6 : RUN bash -c 'touch /app.jar'
 ---> Running in 140f4cff216f
```

```
 ---> 1487e388fc84
Removing intermediate container 140f4cff216f
Step 5/6 : EXPOSE 8761
 ---> Running in db272ee4f5db
 ---> c39508599bf1
Removing intermediate container db272ee4f5db
Step 6/6 : ENTRYPOINT java -Djava.security.egd=file:/dev/./urandom -jar /app.jar
 ---> Running in cd2798828a4f
 ---> d00aad554d2b
Removing intermediate container cd2798828a4f
Successfully built d00aad554d2b
```

由上，可看到镜像构建的详细过程与结果。

测试

1. 启动镜像

```
docker run -d -p 8761:8761 itmuch/microservice-discovery-eureka:0.0.1
```

2. 访问`http://Docker宿主机IP:8761/`，可正常显示 Eureka Server 首页。

可使用相同的方式，将其他微服务也构建成 Docker 镜像。

13.2 使用 Docker Registry 管理 Docker 镜像

至此，已经构建了 Docker 镜像，并将微服务运行在 Docker 之上。但是，一个完整的应用系统可能包含上百个微服务，那就可能对应着上百个镜像，如果考虑各个微服务的版本，那么可能会构建更多的镜像。这些镜像该如何管理呢？

13.2.1 使用 Docker Hub 管理镜像

Docker Hub 是 Docker 官方维护的 Docker Registry，上面存放着很多优秀的镜像。不仅如此，Docker Hub 还提供认证、工作组结构、工作流工具、构建触发器等工具来简化工作。

前文已经讲过，可使用`docker search`命令搜索存放在 Docker Hub 中的镜像。本节将详细探讨 Docker Hub 的使用。

注册与登录

Docker Hub 的使用非常简单，只须注册一个 Docker Hub 账号，就可正常使用了。登录后，可看到 Docker Hub 的主页，如图 13-2 所示。

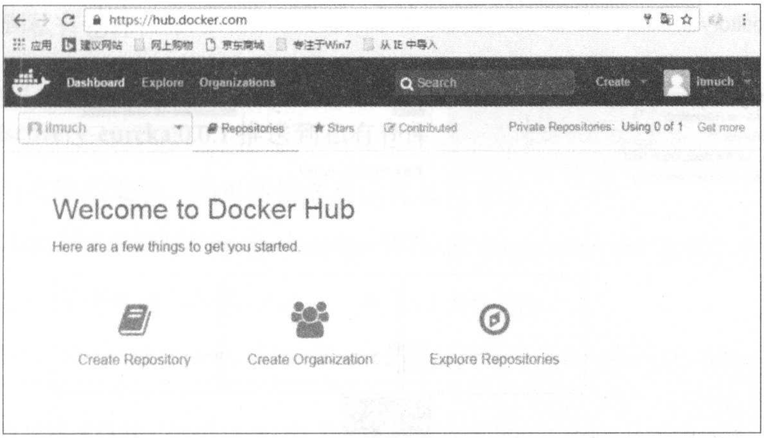


图 13-2 Docker Hub 主页

也可使用`docker login`命令登录 Docker Hub。输入该命令并按照提示输入账号和密码，即可完成登录。例如：

```
$ docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't
have a Docker ID, head over to https://hub.docker.com to create one.
Username: itmuch
Password:
Login Succeeded
```

创建仓库

点击 Docker Hub 主页上的 Create Repository 按钮，按照提示填入信息即可创建一个仓库。如图 13-3 所示，只须填入相关信息，并单击 Create 按钮，就可创建一个名为 microservice-discovery-eureka 的公共仓库。

推送镜像

下面来将前文构建的镜像推送到 Docker Hub。使用以下命令即可，例如：

```
docker push itmuch/microservice-discovery-eureka:0.0.1
```

经过一段时间的等待，就可推送成功。这样，就可在 Docker Hub 查看已推送的镜像。

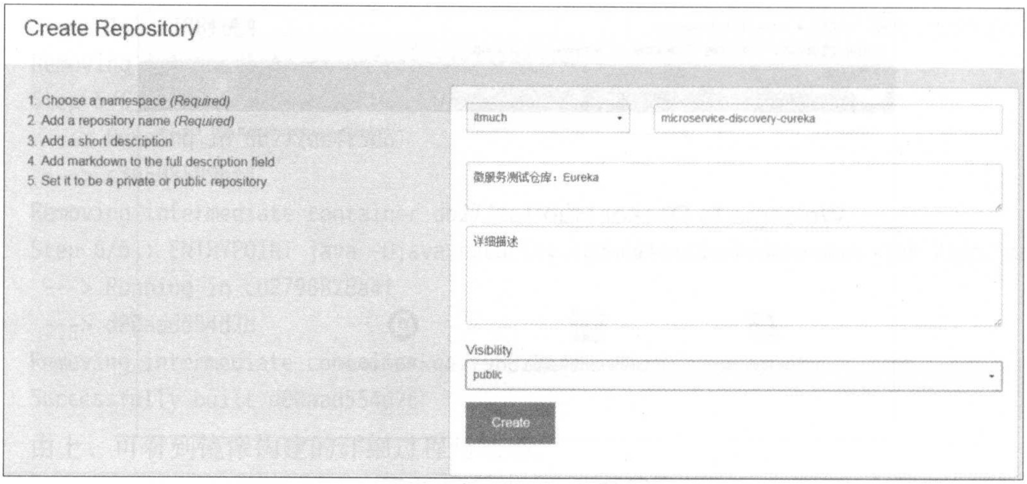


图 13-3 创建仓库界面

13.2.2 使用私有仓库管理镜像

很多场景下，需使用私有仓库管理Docker镜像。相比Docker Hub，私有仓库有以下优势：

- 节省带宽，对于私有仓库中已有的镜像，无须从 Docker Hub 下载，只须从私有仓库中下载即可。
- 更加安全。
- 便于内部镜像的统一管理。

本节来探讨如何搭建、使用私有仓库。可使用 docker-registry 项目或者 Docker Registry 2.0 来搭建私有仓库，但 docker-registry 已被官方标记为过时，并且已有 2 年不维护了，不建议使用。

先用 Docker Registry 2.0 搭建一个私有仓库，然后将 Docker 镜像推送到私有仓库。

搭建私有仓库

Docker Registry 2.0 的搭建非常简单，只须执行以下命令即可新建并启动一个 Docker Registry 2.0。

```
docker run -d -p 5000:5000 --restart=always --name registry2 registry:2
```

将镜像推送到私有仓库

前文使用了`docker push`命令将镜像推送到了 Docker Hub，现在将前文构建的 itmuch/microservice-discovery-eureka:0.0.1 推送到私有仓库。

只须指定私有仓库的地址，即可将镜像推送到私有仓库。

```
docker push localhost:5000/itmuch/microservice-discovery-eureka:0.0.1
```

执行以上命令，发现推送并没有成功，且提示以下内容：

```
The push refers to a repository [localhost:5000/itmuch/microservice-discovery-
    eureka]
An image does not exist locally with the tag: localhost:5000/itmuch/microservice-
    discovery-eureka
```

Docker Hub 是默认的 Docker Registry，所以，itmuch/microservice-discovery-eureka:0.0.1 相当于 docker.io/itmuch/microservice-discovery-eureka:0.0.1。因此，要想将镜像推送到私有仓库，需要修改镜像标签，命令如下：

```
docker tag itmuch/microservice-discovery-eureka:0.0.1 localhost:5000/itmuch/
    microservice-discovery-eureka:0.0.1
```

修改镜像标签后，再次执行以下命令，即可将镜像推送到私有仓库。

```
docker push localhost:5000/itmuch/microservice-discovery-eureka:0.0.1
```

- docker-registry 的 GitHub：*https://github.com/docker/docker-registry*。
- Docker Registry 2.0 的 GitHub：*https://github.com/docker/distribution*。
- 本节中“私有仓库”表示私有 Docker Registry，并非 Docker 中仓库的概念。
- Docker Registry 2.0 需要 Docker 版本高于 1.6.0。
- 还可为私有仓库配置域名、SSL 登录、认证等。限于篇幅，本书不再赘述。有兴趣的读者可参考笔者的开源书：*http://git.oschina.net/itmuch/spring-cloud-book*。
- Docker Registry 2.0 能够满足大部分场景下的需求，但它不包含界面、用户管理、权限控制等功能。如果想要使用这些功能，可使用 Docker Trusted Registry。

13.3 使用 Maven 插件构建 Docker 镜像

Maven 是一个强大的项目管理与构建工具。如果可以使用 Maven 构建 Docker 镜像，工作就能得到进一步的简化。

经过调研，以下几款 Maven 的 Docker 插件进入笔者视野，如表 13-1 所示。

表 13-1 Maven 的 Docker 插件列表

插件名称	官方地址
docker-maven-plugin	https://github.com/spotify/docker-maven-plugin
docker-maven-plugin	https://github.com/fabric8io/docker-maven-plugin
docker-maven-plugin	https://github.com/bibryam/docker-maven-plugin

笔者从各项目的功能性、文档易用性、更新频率、社区活跃度、Stars 等几个纬度考虑，选用了第一款。这是一款由 Spotify 公司开发的 Maven 插件。

下面来详细探讨如何使用 Maven 插件构建 Docker 镜像。

13.3.1 快速入门

以项目microservice-discovery-eureka为例。

1. 在 pom.xml 中添加 Maven 的 Docker 插件。

```
<plugin>
  <groupId>com.spotify</groupId>
  <artifactId>docker-maven-plugin</artifactId>
  <version>0.4.13</version>
  <configuration>
    <imageName>itmuch/microservice-discovery-eureka:0.0.1</imageName>
    <baseImage>java</baseImage>
    <entryPoint>["java", "-jar", "/${project.build.finalName}.jar"]</entryPoint>
    <resources>
      <resource>
        <targetPath>/</targetPath>
        <directory>${project.build.directory}</directory>
        <include>${project.build.finalName}.jar</include>
      </resource>
    </resources>
```

```
  </configuration>
</plugin>
```

简要说明一下插件的配置。

- imageName：用于指定镜像名称，其中 itmuch 是仓库名称，microservice-discovery-eureka 是镜像名称，0.0.1 是标签名称。
- baseImage：用于指定基础镜像，类似于 Dockerfile 中的 FROM 指令。
- entrypoint：类似于 Dockerfile 的 ENTRYPOINT 指令。
- resources.resource.directory：用于指定需要复制的根目录，${project.build.directory} 表示 target 目录。
- resources.resource.include：用于指定需要复制的文件。${project.build.finalName}.jar 指的是打包后的 jar 包文件。

2. 执行以下命令，构建 Docker 镜像。

```
mvn clean package docker:build
```

发现终端输出类似于如下的内容：

```
[INFO] Building image itmuch/microservice-discovery-eureka:0.0.1
Step 1 : FROM java
 ---> 861e95c114d6
Step 2 : ADD /microservice-discovery-eureka-0.0.1-SNAPSHOT.jar //
 ---> 035a03f5b389
Removing intermediate container 2b0e70056f1d
Step 3 : ENTRYPOINT java -jar /microservice-discovery-eureka-0.0.1-SNAPSHOT.jar
 ---> Running in a0149704b949
 ---> eb96ca1402aa
Removing intermediate container a0149704b949
Successfully built eb96ca1402aa
```

由以上日志可知，已成功构建了一个 Docker 镜像。

3. 执行 docker images 命令，即可查看刚刚构建的镜像。

```
REPOSITORY                           TAG   IMAGE ID     CREATED        SIZE
itmuch/microservice-discovery-eureka 0.0.1 eb96ca1402aa 39 seconds ago 685 MB
```

4. 启动以下镜像：

```
docker run -d -p 8761:8761 itmuch/microservice-discovery-eureka:0.0.1
```

发现该 Docker 镜像会很快地启动。

5. 访问测试

   访问 http://Docker 宿主机 IP:8761，能够看到 Eureka Server 的首页。

13.3.2 插件读取 Dockerfile 进行构建

之前的示例直接在 pom.xml 中设置了一些构建的参数。很多场景下希望使用 Dockerfile 更精确、有可读性地构建镜像。

1. 首先在/microservice-discovery-eureka/src/main/docker 目录下，新建一个 Dockerfile 文件，例如：

```
FROM java:8
VOLUME /tmp
ADD microservice-discovery-eureka-0.0.1-SNAPSHOT.jar app.jar
RUN bash -c 'touch /app.jar'
EXPOSE 9000
ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]
```

2. 修改 pom.xml：

```
<plugin>
  <groupId>com.spotify</groupId>
  <artifactId>docker-maven-plugin</artifactId>
  <version>0.4.13</version>
  <configuration>
    <imageName>itmuch/microservice-discovery-eureka:0.0.2</imageName>
    <dockerDirectory>${project.basedir}/src/main/docker</dockerDirectory>
    <resources>
      <resource>
        <targetPath>/</targetPath>
        <directory>${project.build.directory}</directory>
        <include>${project.build.finalName}.jar</include>
      </resource>
    </resources>
  </configuration>
</plugin>
```

   可以看到，不再指定 baseImage 和 entrypoint，而是使用 dockerDirectory 指定 Dockerfile 所在的路径。这样，就可以使用 Dockerfile 构建 Docker 镜像了。

13.3.3　将插件绑定在某个 phase 执行

很多场景下，有这样的需求，执行例如mvn clean package时，插件就自动为构建 Docker 镜像。要想实现这点，只须将插件的 goal 绑定在某个 phase 即可。

phase 和 goal 可以这样理解：maven 命令格式是：mvn phase:goal，例如mvn package docker :build。那么，package 和 docker 都是 phase，build 则是 goal。示例：

```
<plugin>
  <groupId>com.spotify</groupId>
  <artifactId>docker-maven-plugin</artifactId>
  <version>0.4.13</version>
  <executions>
    <execution>
      <id>build-image</id>
      <phase>package</phase>
      <goals>
        <goal>build</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <imageName>itmuch/microservice-discovery-eureka:0.0.3</imageName>
    <baseImage>java</baseImage>
    <entryPoint>["java", "-jar", "/${project.build.finalName}.jar"]</entryPoint>
    <resources>
      <resource>
        <targetPath>/</targetPath>
        <directory>${project.build.directory}</directory>
        <include>${project.build.finalName}.jar</include>
      </resource>
    </resources>
  </configuration>
</plugin>
```

由配置可知，只须添加如下配置：

```
<executions>
  <execution>
    <id>build-image</id>
    <phase>package</phase>
```

```
    <goals>
      <goal>build</goal>
    </goals>
  </execution>
</executions>
```

就可将插件绑定在 package 这个 phase 上。也就是说，用户只须执行`mvn package`，就会自动执行`mvn docker:build`。当然，读者也可按照需求，将插件绑定到其他的 phase。

13.3.4 推送镜像

前文使用`docker push` 命令实现了镜像的推送，也可使用 Maven 插件推送镜像。不妨使用 Maven 插件推送一个 Docker 镜像到 Docker Hub。

1. 修改 Maven 的全局配置文件 setttings.xml，在其中添加以下内容，配置 Docker Hub 的用户信息。

```
<server>
  <id>docker-hub</id>
  <username>你的DockerHub用户名</username>
  <password>你的DockerHub密码</password>
  <configuration>
    <email>你的DockerHub邮箱</email>
  </configuration>
</server>
```

2. 修改 pom.xml，示例：

```
<plugin>
  <groupId>com.spotify</groupId>
  <artifactId>docker-maven-plugin</artifactId>
  <version>0.4.13</version>
  <configuration>
    <imageName>itmuch/microservice-discovery-eureka:0.0.4</imageName>
    <baseImage>java</baseImage>
    <entryPoint>["java", "-jar", "/${project.build.finalName}.jar"]</entryPoint>
    <resources>
      <resource>
        <targetPath>/</targetPath>
        <directory>${project.build.directory}</directory>
        <include>${project.build.finalName}.jar</include>
      </resource>
```

```
    </resources>

    <!-- 与maven配置文件settings.xml中配置的server.id一致，用于推送镜像 -->
    <serverId>docker-hub</serverId>
  </configuration>
</plugin>
```

如上，添加 serverId 段落，并引用 settings.xml 中的 server 的 id 即可。

3. 执行以下命令，添加 pushImage 的标识，表示推送镜像。

```
mvn clean package docker:build  -DpushImage
```

经过一段时间的等待，会发现 Docker 镜像已经被 push 到 Docker Hub 了。同理，也可推送镜像到私有仓库，只需要将 imageName 指定成类似于如下的形式即可：

```
<imageName>localhost:5000/itmuch/microservice-discovery-eureka:0.0.4</imageName>
```

- 以上示例中，是通过 imageName 指定镜像名称和标签的，例如：

```
<imageName>itmuch/microservice-discovery-eureka:0.0.4</imageName>
```

  也可借助 imageTags 元素更为灵活地指定镜像名称和标签，例如：

```
<configuration>
  <imageName>itmuch/microservice-discovery-eureka</imageName>
  <imageTags>
    <imageTag>0.0.5</imageTag>
    <imageTag>latest</imageTag>
  </imageTags>
  ...
<configuration>
```

  这样就可为同一个镜像指定两个标签。

- 也可在执行构建命令时，使用 dockerImageTags 参数指定标签名称，例如：

```
mvn clean package docker:build -DpushImageTags -DdockerImageTags=
latest -DdockerImageTags=another-tag
```

- 如需重复构建相同标签名称的镜像，可将 forceTags 设为 true，这样就会覆盖构建相同标签的镜像。

```
<configuration>
  <!-- optionally overwrite tags every time image is built with
      docker:build -->
  <forceTags>true</forceTags>
<configuration>
```

Spotify 是全球最大的正版流媒体音乐服务平台。

13.4 常见问题与总结

Docker 官方说明文档非常完备，其中对 Docker 的常见问题进行了详细的总结。详见：*https://docs.docker.com/engine/faq/*。

14 使用 Docker Compose 编排微服务

经过前文讲解，可使用 Dockerfile（或 Maven）构建镜像，然后使用 docker 命令操作容器，例如 docker run、docker kill 等。然而，使用微服务架构的应用系统一般包含若干个微服务，每个微服务一般都会部署多个实例。如果每个微服务都要手动启停，那么效率之低、维护量之大可想而知。

本章将讨论如何使用 Docker Compose 来轻松、高效地管理容器。为了简单起见，本章将 Docker Compose 简称为 Compose。

14.1 Docker Compose 简介

Compose 是一个用于定义和运行多容器 Docker 应用程序的工具，前身是 Fig。它非常适合用在开发、测试、构建 CI 工作流等场景。本书所使用的 Compose 版本是 1.10.0。

> Compose 的 GitHub：*https://github.com/docker/compose*。

14.2 安装 Docker Compose

本节来讨论如何安装 Compose。

14.2.1 安装 Compose

Compose 有多种安装方式，例如通过 Shell、pip 以及将 Compose 作为容器安装等。本书讲解通过 Shell 来安装的方式，其他安装方式可详见官方文档：*https://docs.docker.com/compose/install/*。

1. 通过以下命令自动下载并安装适应系统版本的 Compose：

```
curl -L "https://github.com/docker/compose/releases/download/1.10.0/docker-
compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
```

2. 为安装脚本添加执行权限：

```
chmod +x /usr/local/bin/docker-compose
```

这样，Compose 就安装完成了。

可使用以下命令测试安装结果：

```
docker-compose --version
```

可输出类似于如下的内容：

```
docker-compose version 1.10.0, build 4bd6f1a
```

说明 Compose 已成功安装。

14.2.2 安装 Compose 命令补全工具

现在已成功安装 Compose，然而，当输入`docker-compose`并按下 Tab 键时，Compose 并没有补全命令。要想使用 Compose 的命令补全，需要安装命令补全工具。

命令补全工具在 Bash 和 Zsh 下的安装方式不同，本书演示的是 Bash 下的安装。其他 Shell 以及其他操作系统上的安装，可详见 Docker 的官方文档：*https://docs.docker.com/compose/completion/*，笔者不再赘述。

执行以下命令，即可安装命令补全工具：

```
curl -L https://raw.githubusercontent.com/docker/compose/$(docker-compose version
--short)/contrib/completion/bash/docker-compose -o /etc/bash_completion.d/docker-
compose
```

这样，在重新登录后，输入`docker-compose`并按下 Tab 键，Compose 就可自动补全命令了。

14.3 Docker Compose 快速入门

本节将探讨 Compose 使用的基本步骤，并编写一个简单示例快速入门。

14.3.1 基本步骤

使用 Compose 大致有 3 个步骤：

- 使用 Dockerfile（或其他方式）定义应用程序环境，以便在任何地方重现该环境。
- 在 docker-compose.yml 文件中定义组成应用程序的服务，以便各个服务在一个隔离的环境中一起运行。
- 运行 docker-compose up 命令，启动并运行整个应用程序。

14.3.2 入门示例

下面以 microservice-discovery-eureka 为例讲解 Compose 的基本步骤。

1. 使用`mvn clean package`命令打包项目，获得 jar 包`microservice-discovery-eureka-0.0.1-SNAPSHOT.jar`。
2. 在`microservice-discovery-eureka-0.0.1-SNAPSHOT.jar`所在路径（默认是项目的 target 目录）创建 Dockerfile 文件，并在其中添加如下内容。

```
FROM java:8
VOLUME /tmp
ADD microservice-discovery-eureka-0.0.1-SNAPSHOT.jar app.jar
RUN bash -c 'touch /app.jar'
EXPOSE 9000
ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]
```

3. 在`microservice-discovery-eureka-0.0.1-SNAPSHOT.jar`所在路径创建文件 docker-compose.yml，在其中添加如下内容。

```
version: '2'          # 表示该docker-compose.yml文件使用的是Version 2 file
  format
services:
  eureka:             # 指定服务名称
    build: .          # 指定Dockerfile所在路径
    ports:
      - "8761:8761" # 指定端口映射，类似docker run的-p选项，注意使用字符串
        形式
```

4. 在docker-compose.yml所在路径执行以下命令：

   ```
   docker-compose up
   ```

   Compose 就会自动构建镜像并使用镜像启动容器。也可使用docker-compose up -d后台启动并运行这些容器。

5. 访问http://宿主机IP:8761/，即可访问 Eureka Server 首页。

14.3.3 工程、服务、容器

Docker Compose 将所管理的容器分为三层，分别是工程（project），服务（service）以及容器（container）。Docker Compose 运行目录下的所有文件（docker-compose.yml、extends 文件或环境变量文件等）组成一个工程（默认为 docker-compose.yml 所在目录的目录名称）。一个工程可包含多个服务，每个服务中定义了容器运行的镜像、参数和依赖，一个服务可包括多个容器实例。

对应 14.3.2 节，工程名称是 docker-compose.yml 所在的目录名。该工程包含了 1 个服务，服务名称是 eureka。执行 docker-compose up 时，启动了 eureka 服务的 1 个容器实例。

14.4 docker-compose.yml 常用命令

docker-compose.yml 是 Compose 的默认模板文件。该文件有多种写法，例如 Version 1 file format、Version 2 file format、Version 2.1 file format、Version 3 file format 等。其中，Version 1 file format 将逐步被被弃用，Version 2.x 及 Version 3.x 基本兼容，是未来的趋势。考虑到目前业界的使用情况，本节只讨论 Version 2 file format 下的常用命令。

- build

  配置构建时的选项，Compose 会利用它自动构建镜像。build 的值可以是一个路径，例如：

  ```
  build: ./dir
  ```

  也可以是一个对象，用于指定 Dockerfile 和参数，例如：

  ```
  build:
    context: ./dir
    dockerfile: Dockerfile-alternate
    args:
      buildno: 1
  ```

- command

  覆盖容器启动后默认执行的命令，示例：

```
command: bundle exec thin -p 3000
```

  也可以是一个 list，类似于 Dockerfile 中的 CMD 指令，格式如下：

```
command: [bundle, exec, thin, -p, 3000]
```

- dns

  配置 dns 服务器。可以是一个值，也可以是一个列表。示例：

```
dns: 8.8.8.8
dns:
  - 8.8.8.8
  - 9.9.9.9
```

- dns_search

  配置 DNS 的搜索域，可以是一个值，也可以是一个列表。示例：

```
dns_search: example.com
dns_search:
  - dc1.example.com
  - dc2.example.com
```

- environment

  环境变量设置，可使用数组或字典两种方式。示例：

```
environment:
  RACK_ENV: development
  SHOW: 'true'
  SESSION_SECRET:

environment:
  - RACK_ENV=development
  - SHOW=true
  - SESSION_SECRET
```

- env_file

  从文件中获取环境变量，可指定一个文件路径或路径列表。如果通过`docker-compose -f FILE`指定了 Compose 文件，那么 env_file 中的路径是 Compose 文件所在目录的相对路径。使用 environment 指定的环境变量会覆盖 env_file 指定的环境变量。示例：

```
env_file: .env

env_file:
  - ./common.env
  - ./apps/web.env
  - /opt/secrets.env
```

- expose

暴露端口，只将端口暴露给连接的服务，而不暴露给宿主机。示例：

```
expose:
 - "3000"
 - "8000"
```

- external_links

连接到 docker-compose.yml 外部的容器，甚至并非 Compose 管理的容器，特别是提供共享或公共服务的容器。格式跟 links 类似，例如：

```
external_links:
 - redis_1
 - project_db_1:mysql
 - project_db_1:postgresql
```

- image

指定镜像名称或镜像 ID，如果本地不存在该镜像，Compose 会尝试下载该镜像。

示例：

```
image: java
```

- links

连接到其他服务的容器。可以指定服务名称和服务别名（SERVICE:ALIAS），也可只指定服务名称。例如：

```
web:
  links:
   - db
   - db:database
   - redis
```

- networks

详见本书 14.6 节。

- network_mode

  设置网络模式。示例：

```
network_mode: "bridge"
network_mode: "host"
network_mode: "none"
network_mode: "service:[service name]"
network_mode: "container:[container name/id]"
```

- ports

  暴露端口信息，可使用HOST:CONTAINER的格式，也可只指定容器端口（此时宿主机将会随机选择端口），类似于docker run -p。

  需要注意的是，当使用HOST:CONTAINER格式映射端口时，容器端口小于 60 将会得到错误的接口，因为 yaml 会把xx:yy的数字解析为 60 进制。因此，建议使用字符串的形式。示例：

```
ports:
 - "3000"
 - "3000-3005"
 - "8000:8000"
 - "9090-9091:8080-8081"
 - "49100:22"
 - "127.0.0.1:8001:8001"
 - "127.0.0.1:5000-5010:5000-5010"
```

- volumes

  卷挂载路径设置。可以设置宿主机路径（HOST:CONTAINER），也可指定访问模式（HOST:CONTAINER:ro）。示例：

```
volumes:
  # Just specify a path and let the Engine create a volume
  - /var/lib/mysql
  # Specify an absolute path mapping
  - /opt/data:/var/lib/mysql
  # Path on the host, relative to the Compose file
  - ./cache:/tmp/cache
  # User-relative path
  - ~/configs:/etc/configs/:ro
  # Named volume
  - datavolume:/var/lib/mysql
```

- volumes_from

  从另一个服务或容器挂载卷。可指定只读（ro）或读写（rw），默认是读写（rw）。示例：

```
volumes_from:
 - service_name
 - service_name:ro
 - container:container_name
 - container:container_name:rw
```

> docker-compose.yml 还有很多其他命令，比如 depends_on、pid、devices 等。限于篇幅，笔者仅挑选常用的命令进行讲解，其他命令不再赘述。感兴趣的读者们可参考官方文档：*https://docs.docker.com/compose/compose-file/*。

14.5 docker-compose 常用命令

和 docker 命令一样，docker-compose 命令也有很多选项。下面来详细探讨 docker-compose 的常用命令。

- build

  构建或重新构建服务。服务被构建后将会以`project_service`的形式标记，例如：`composetest_db`。

- help

  查看指定命令的帮助文档，该命令非常实用。docker-compose 所有命令的帮助文档都可通过该命令查看。

```
docker-compose help COMMAND
```

  示例：

```
docker-compose help build        # 查看docker-compose build的帮助
```

- kill

  通过发送`SIGKILL`信号停止指定服务的容器。示例：

```
docker-compose kill eureka
```

  该命令也支持通过参数来指定发送的信号，例如：

```
docker-compose kill -s SIGINT
```

- logs

  查看服务的日志输出。

- port

  打印绑定的公共端口。示例：

  ```
  docker-compose port eureka 8761
  ```

  这样就可输出 eureka 服务 8761 端口所绑定的公共端口。

- ps

  列出所有容器。示例：

  ```
  docker-compose ps
  ```

  也可列出指定服务的容器，示例：

  ```
  docker-compose ps eureka
  ```

- pull

  下载服务镜像。

- rm

  删除指定服务的容器。示例：

  ```
  docker-compose rm eureka
  ```

- run

  在一个服务上执行一个命令。示例：

  ```
  docker-compose run web bash
  ```

  这样即可启动一个 web 服务，同时执行 bash 命令。

- scale

  设置指定服务运行容器的个数，以 service=num 的形式指定。示例：

  ```
  docker-compose scale user=3 movie=3
  ```

- start

  启动指定服务已存在的容器。示例：

  ```
  docker-compose start eureka
  ```

- stop

  停止已运行的容器。示例：

  ```
  docker-compose stop eureka
  ```

停止后，可使用`docker-compose start`再次启动这些容器。

- up

  构建、创建、重新创建、启动，连接服务的相关容器。所有连接的服务都会启动，除非它们已经运行。

  `docker-compose up`命令会聚合所有容器的输出，当命令退出时，所有容器都会停止。

  使用`docker-compose up -d`可在后台启动并运行所有容器。

> 本节仅讨论常用的 docker-compose 命令，其他命令可详见 Docker 官方文档：*https://docs.docker.com/compose/reference/overview/*。

14.6 Docker Compose 网络设置

本节将详细探讨 Compose 的网络设置。本节介绍的网络特性仅适用于 Version 2 file format，Version 1 file format 不支持该特性。

14.6.1 基本概念

默认情况下，Compose 会为应用创建一个网络，服务的每个容器都会加入该网络中。这样，容器就可被该网络中的其他容器访问，不仅如此，该容器还能以服务名称作为 hostname 被其他容器访问。

默认情况下，应用程序的网络名称基于 Compose 的工程名称，而项目名称基于 docker-compose.yml 所在目录的名称。如需修改工程名称，可使用--project-name 标识或 COMPOSE_PORJECT_NAME 环境变量。

举个例子，假如一个应用程序在名为 myapp 的目录中，并且 docker-compose.yml 如下所示：

```
version: '2'
services:
  web:
    build: .
    ports:
      - "8000:8000"
  db:
    image: postgres
```

当运行 docker-compose up 时，将会执行以下几步：

1. 创建一个名为 myapp_default 的网络。
2. 使用 web 服务的配置创建容器，它以“web”这个名称加入网络 myapp_default。
3. 使用 db 服务的配置创建容器，它以“db”这个名称加入网络 myapp_default。

容器间可使用服务名称（web 或 db）作为 hostname 相互访问。例如，web 这个服务可使用`postgres://db:5432` 访问 db 容器。

14.6.2 更新容器

当服务的配置发生更改时，可使用 docker-compose up 命令更新配置。

此时，Compose 会删除旧容器并创建新容器。新容器会以不同的 IP 地址加入网络，名称保持不变。任何指向旧容器的连接都会被关闭，容器会重新找到新容器并连接上去。

14.6.3 links

前文讲过，默认情况下，服务之间可使用服务名称相互访问。links 允许定义一个别名，从而使用该别名访问其他服务。举个例子：

```
version: '2'
services:
  web:
    build: .
    links:
      - "db:database"
  db:
    image: postgres
```

这样 Web 服务就可使用 db 或 database 作为 hostname 访问 db 服务了。

14.6.4 指定自定义网络

一些场景下，默认的网络配置满足不了我们的需求，此时可使用 networks 命令自定义网络。networks 命令允许创建更加复杂的网络拓扑并指定自定义网络驱动和选项。不仅如此，还可使用 networks 将服务连接到不是由 Compose 管理的、外部创建的网络。

如下，在其中定义了两个自定义网络。

```
version: '2'

services:
```

```
  proxy:
    build: ./proxy
    networks:
      - front
  app:
    build: ./app
    networks:
      - front
      - back
  db:
    image: postgres
    networks:
      - back

networks:
  front:
    # Use a custom driver
    driver: custom-driver-1
  back:
    # Use a custom driver which takes special options
    driver: custom-driver-2
    driver_opts:
      foo: "1"
      bar: "2"
```

其中，proxy 服务与 db 服务隔离，两者分别使用自己的网络，app 服务可与两者通信。由本例不难发现，使用 networks 命令，即可方便实现服务间的网络隔离与连接。

14.6.5 配置默认网络

除自定义网络外，也可为默认网络自定义配置。

```
version: '2'

services:
  web:
    build: .
    ports:
      - "8000:8000"
  db:
```

```
    image: postgres

networks:
  default:
    # Use a custom driver
    driver: custom-driver-1
```

这样，就可为该应用指定自定义的网络驱动。

14.6.6 使用已存在的网络

一些场景下，并不需要创建新的网络，而只须加入已存在的网络，此时可使用 external 选项。示例：

```
networks:
  default:
    external:
      name: my-pre-existing-network
```

14.7 综合实战：使用 Docker Comose 编排 Spring Cloud 微服务

本节将使用 Compose 编排前文编写的 Spring Cloud 微服务。

14.7.1 编排 Spring Cloud 微服务

本节来编排前文编写的 Spring Cloud 微服务。表 14-1 是本节编排时用到的微服务项目。

表 14-1 本节编排的微服务列表

微服务项目名称	项目微服务中的角色
microservice-discovery-eureka	服务发现组件
microservice-provider-user	服务提供者
microservice-consumer-movie-ribbon-hystrix	服务消费者
microservice-gateway-zuul	API Gateway
microservice-hystrix-turbine	Hystrix 聚合监控工具
microservice-hystrix-dashboard	Hystrix 监控界面

编写代码

1. 本节使用 Maven 插件构建 Docker 镜像，在各个项目的 pom.xml 中添加以下内容。

```
<!-- 添加docker-maven插件 -->
<plugin>
  <groupId>com.spotify</groupId>
  <artifactId>docker-maven-plugin</artifactId>
  <version>0.4.13</version>
  <configuration>
    <imageName>itmuch/${project.artifactId}:${project.version}</imageName>
    <forceTags>true</forceTags>
    <baseImage>java</baseImage>
    <entryPoint>["java", "-jar", "/${project.build.finalName}.jar"]</entryPoint>
    <resources>
      <resource>
        <targetPath>/</targetPath>
        <directory>${project.build.directory}</directory>
        <include>${project.build.finalName}.jar</include>
      </resource>
    </resources>
  </configuration>
</plugin>
```

   由配置可知，构建出来的镜像名称是itmuch/各个微服务的artifactId:各个微服务的版本，例如：microservice-discovery-eureka:0.0.1-SNAPSHOT。

2. 前文中为各个项目配置的eureka.client.serviceUrl.defaultZone的值是*http://localhost:8761/eureka/*。由于 Docker 默认的网络模式是 bridge，各个容器的 IP 都不相同，因此使用*http://localhost:8761/eureka/*满足不了需求。可为 Eureka Server 所在容器配置一个主机名（例如 discover），并让各个微服务使用主机名访问 Eureka Server。

   将所有微服务eureka.client.serviceUrl.defaultZone修改为如下内容。

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://discovery:8761/eureka/
```

3. 在每个项目的根目录执行以下命令，构建 Docker 镜像。

```
mvn clean package docker:build
```

4. 编写 docker-compose.yml。

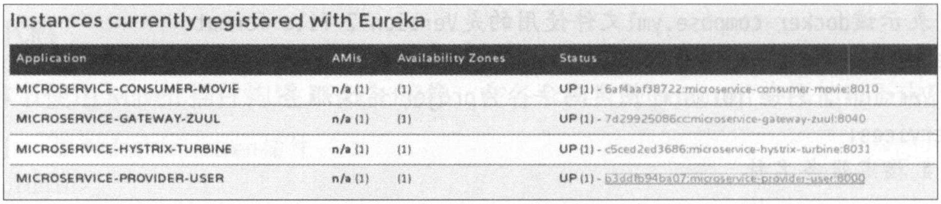
```
# 表示该docker-compose.yml文件使用的是Version 2 file format
version: '2'
# Version 2 file format的固定写法，为project定义服务
services:
  # 指定服务名称
  microservice-discovery-eureka:
    # 指定服务所使用的镜像
    image: itmuch/microservice-discovery-eureka:0.0.1-SNAPSHOT
    # 暴露端口信息
    ports:
      - "8761:8761"
  microservice-provider-user:
    image: itmuch/microservice-provider-user:0.0.1-SNAPSHOT
    # 连接到microservice-discovery-eureka，这边使用的是SERVICE:ALIAS的形式
    links:
      - microservice-discovery-eureka:discovery
  microservice-consumer-movie-ribbon-hystrix:
    image: itmuch/microservice-consumer-movie-ribbon-hystrix:0.0.1-SNAPSHOT
    links:
      - microservice-discovery-eureka:discovery
  microservice-gateway-zuul:
    image: itmuch/microservice-gateway-zuul:0.0.1-SNAPSHOT
    links:
      - microservice-discovery-eureka:discovery
  microservice-hystrix-dashboard:
    image: itmuch/microservice-hystrix-dashboard:0.0.1-SNAPSHOT
    links:
      - microservice-discovery-eureka:discovery
  microservice-hystrix-turbine:
    image: itmuch/microservice-hystrix-turbine:0.0.1-SNAPSHOT
    links:
      - microservice-discovery-eureka:discovery
```

启动与测试

1. 执行以下命令启动项目。

```
docker-compose up
```

效果如图 14-1 所示。


Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
MICROSERVICE-CONSUMER-MOVIE	n/a (1)	(1)	UP (1) - 6af4aaf38722:microservice-consumer-movie:8010
MICROSERVICE-GATEWAY-ZUUL	n/a (1)	(1)	UP (1) - 7d29925086cc:microservice-gateway-zuul:8040
MICROSERVICE-HYSTRIX-TURBINE	n/a (1)	(1)	UP (1) - c5ced2ed3686:microservice-hystrix-turbine:8031
MICROSERVICE-PROVIDER-USER	n/a (1)	(1)	UP (1) - b3ddfb94ba07:microservice-provider-user:8000

图 14-1　Eureka Server 上的微服务列表

2. 按照前文章节中的描述，测试各微服务是否能够正常运行。

简化 Compose 编写

前文讲过，在 Version 2 file format 的 docker-compose.yml 中，同一个 Compose 工程中的所有服务共享一个隔离网络，可使用服务名称作为主机名来发现其他服务。因此，本节中的 docker-compose.yml 也可简化成如下形式。

```
version: '2'
services:
  discovery:
    image: itmuch/microservice-discovery-eureka:0.0.1-SNAPSHOT
    ports:
      - "8761:8761"
  microservice-provider-user:
    image: itmuch/microservice-provider-user:0.0.1-SNAPSHOT
  microservice-consumer-movie-ribbon-hystrix:
    image: itmuch/microservice-consumer-movie-ribbon-hystrix:0.0.1-SNAPSHOT
  microservice-gateway-zuul:
    image: itmuch/microservice-gateway-zuul:0.0.1-SNAPSHOT
  microservice-hystrix-dashboard:
    image: itmuch/microservice-hystrix-dashboard:0.0.1-SNAPSHOT
  microservice-hystrix-turbine:
    image: itmuch/microservice-hystrix-turbine:0.0.1-SNAPSHOT
```

> Version 2 file format 与 Version 1 file format 的区别造成的问题：*https://github.com/docker/compose/issues/3926*。

14.7.2 编排高可用的 Eureka Server

在 14.7.2 节中，构建了一个双节点的 Eureka Server 集群，本节将使用 Compose 编排该 Eureka Server 集群。

1. 执行以下命令构建 Docker 镜像。

```
mvn clean package docker:build
```

2. 编写 docker-compose.yml，如下：

```
version: "2"            # 表示使用docker-compose.yml的Version 2 file format编写
services:
 microservice-discovery-eureka-ha1:
   hostname: peer1             # 指定hostname
   image: itmuch/microservice-discovery-eureka-ha:0.0.1-SNAPSHOT
   links:
     - microservice-discovery-eureka-ha2
   ports:
     - "8761:8761"
   environment:
     - spring.profiles.active=peer1
 microservice-discovery-eureka-ha2:
   hostname: peer2
   image: itmuch/microservice-discovery-eureka-ha:0.0.1-SNAPSHOT
   links:
     - microservice-discovery-eureka-ha1
   ports:
     - "8762:8762"
   environment:
     - spring.profiles.active=peer2
```

由文件内容可知，我们定义了两个服务：microservice-discovery-eureka-ha1 和 microservice-discovery-eureka-ha2。他们的 hostname 分别是 peer1 和 peer2，通过 links 标签互相连接。

3. 执行以下命令启动项目。

```
docker-compose up
```

然而，终端会输出类似以下的异常：

```
ERROR: Circular dependency between microservice-discovery-eureka-ha1 and
microservice-discovery-eureka-ha2
```

从异常可知，该写法存在循环依赖。也就是说，links 无法实现双向连接。如何解决这个问题呢？

解决循环依赖

该问题有很多解决方案，例如使用 ambassador pattern，使用外部 DNS 容器等。本节采用最简单的方式，配置如下。

```
version: "2"
services:
  # 默认情况下，其他服务可使用服务名称连接到该服务。对于peer2节点，它需连接
  # http://peer1:8761/eureka/，因此，可配置该服务的名称为peer1
  peer1:
    image: itmuch/microservice-discovery-eureka-ha:0.0.1-SNAPSHOT
    ports:
      - "8761:8761"
    environment:
      - spring.profiles.active=peer1
  peer2:
    image: itmuch/microservice-discovery-eureka-ha:0.0.1-SNAPSHOT
    hostname: peer2
    ports:
      - "8762:8762"
    environment:
      - spring.profiles.active=peer2
```

- 解决循环依赖的总结：*http://www.dockone.io/article/929*。
- ambassador pattern 官方介绍：*https://docs.docker.com/engine/admin/ambassador_pattern_linking/*。
- StackOverflow 上对该问题的深入探讨：*http://stackoverflow.com/questions/29307645/how-to-link-docker-container-to-each-other-with-docker-compose*。
- Github 上的相关 Issue：*https://github.com/docker/compose/issues/666*。

14.7.3 编排高可用 Spring Cloud 微服务集群及动态伸缩

本节再来写一个 Compose 编排 Spring Cloud 微服务的示例。在这个示例中，所有微服务节点最终都是高可用的。表 14-2 是本节编排时使用到的微服务项目。

表 14-2　本节编排的微服务列表

微服务项目名称	项目微服务中的角色
microservice-discovery-eureka-ha	服务发现组件
microservice-provider-user	服务提供者
microservice-consumer-movie-ribbon-hystrix	服务消费者
microservice-gateway-zuul	API Gateway
microservice-hystrix-turbine	Hystrix 聚合监控工具

编写代码

1. 由于使用了microservice-discovery-eureka-ha，需要将所有微服务的eureka.client.serviceUrl.defaultZone 属性修改为如下内容。

```
eureka:
  client:
    service-url:
      defaultZone: http://peer1:8761/eureka/,http://peer2:8762/eureka/
```

2. 在每个项目的根目录，执行以下命令构建 Docker 镜像。

```
mvn clean package docker:build
```

3. 编写 docker-compose.yml。

```
version: "2"
services:
  peer1:
    image: itmuch/microservice-discovery-eureka-ha:0.0.1-SNAPSHOT
    ports:
      - "8761:8761"
    environment:
      - spring.profiles.active=peer1
  peer2:
    image: itmuch/microservice-discovery-eureka-ha:0.0.1-SNAPSHOT
    hostname: peer2
    ports:
      - "8762:8762"
    environment:
      - spring.profiles.active=peer2
```

```
microservice-provider-user:
  image: itmuch/microservice-provider-user:0.0.1-SNAPSHOT
microservice-consumer-movie-ribbon-hystrix:
  image: itmuch/microservice-consumer-movie-ribbon-hystrix:0.0.1-SNAPSHOT
microservice-gateway-zuul:
  image: itmuch/microservice-gateway-zuul:0.0.1-SNAPSHOT
microservice-hystrix-turbine:
  image: itmuch/microservice-hystrix-turbine:0.0.1-SNAPSHOT
```

启动与测试

1. 执行以下命令启动项目

```
docker-compose up
```

启动后的效果如图 14-2 所示。

Application	AMIs	Availability Zones	Status
MICROSERVICE-CONSUMER-MOVIE	n/a (1)	(1)	UP (1) - 9d5a2bb10ced:microservice-consumer-movie:8010
MICROSERVICE-DISCOVERY-EUREKA-HA	n/a (2)	(2)	UP (2) - f33628b79bbb:microservice-discovery-eureka-ha:8761 , peer2:microservice-discovery-eureka-ha:8762
MICROSERVICE-GATEWAY-ZUUL	n/a (1)	(1)	UP (1) - ec48c421f515:microservice-gateway-zuul:8040
MICROSERVICE-HYSTRIX-TURBINE	n/a (1)	(1)	UP (1) - 933c76c01717:microservice-hystrix-turbine:8031
MICROSERVICE-PROVIDER-USER	n/a (1)	(1)	UP (1) - 31dcd80e2421:microservice-provider-user:8000

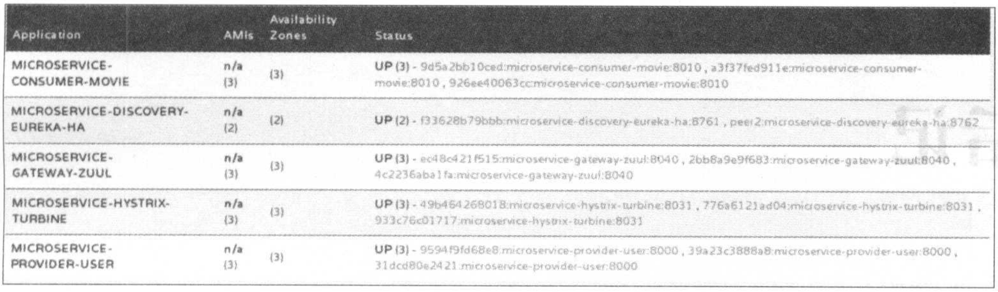
图 14-2　Eureka Server 上的微服务列表

2. 按照前文各章节的讲解，测试各微服务能否正常运行。
3. 执行以下命令，为各个微服务动态扩容。让除 Eureka Server 以外的所有微服务都启动 3 个实例。

```
docker-compose scale microservice-provider-user=3 microservice-consumer-movie-ribbon-hystrix=3 microservice-gateway-zuul=3 microservice-hystrix-turbine=3
```

效果如图 14-3 所示。

由图可知，除 Eureka Server 外，每个微服务都启动了 3 个节点，说明已实现微服务的动态扩容。同理，也可实现动态缩容。



Application	AMIs	Availability Zones	Status
MICROSERVICE-CONSUMER-MOVIE	n/a (3)	(3)	UP (3) - 9d5a2bb10ced:microservice-consumer-movie:8010 , a3f37fed911e:microservice-consumer-movie:8010 , 926ee40063cc:microservice-consumer-movie:8010
MICROSERVICE-DISCOVERY-EUREKA-HA	n/a (2)	(2)	UP (2) - f33628b79bbb:microservice-discovery-eureka-ha:8761 , peer2:microservice-discovery-eureka-ha:8762
MICROSERVICE-GATEWAY-ZUUL	n/a (3)	(3)	UP (3) - ec48c421f515:microservice-gateway-zuul:8040 , 2bb8a9e9f683:microservice-gateway-zuul:8040 , 4c2236aba1fa:microservice-gateway-zuul:8040
MICROSERVICE-HYSTRIX-TURBINE	n/a (3)	(3)	UP (3) - 49b464268018:microservice-hystrix-turbine:8031 , 776a6121ad04:microservice-hystrix-turbine:8031 , 933c76c01717:microservice-hystrix-turbine:8031
MICROSERVICE-PROVIDER-USER	n/a (3)	(3)	UP (3) - 9594f9fd68e8:microservice-provider-user:8000 , 39a23c3888a8:microservice-provider-user:8000 , 31dcd80e2421:microservice-provider-user:8000

图 14-3 Eureka Server 上的微服务列表

14.8 常见问题与总结

Compose 官方说明文档非常完备，其中对 Docker 的常见问题进行了详细的总结。详见：*https://docs.docker.com/compose/faq/*。

后记

至此，Spring Cloud 与 Docker 微服务架构实战的探索之旅已经结束。

由于篇幅有限，笔者无法为大家讲述微服务中的方方面面。微服务是一个非常宏观的话题，要想切实落地微服务架构，光靠一两本书是远远不够的。微服务粒度、持续集成、自动化机制、组织机构的建设乃至如何从传统架构向微服务架构迁移，都是值得我们深思的问题。

所幸，Pivotal 工程师 Matt Stine 的著作 *Migrating to Cloud-Native Application Architectures*（迁移到云原生应用架构）对本书未提到的许多话题有非常精辟的阐述，相信大家阅读后会对微服务有更深的理解与体会。

最后，希望大家能够享受这段 Spring Cloud 与 Docker 实战微服务架构的旅行，也希望本书可以切实地帮助大家实现微服务架构的落地。

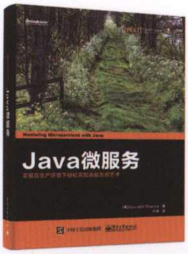
> *Migrating to Cloud-Native Application Architectures*（迁移到云原生应用架构）阅读地址：*https://pivotal.io/platform-as-a-service/migrating-to-cloud-native-application-architectures-ebook*，也可使用搜索引擎搜索中文翻译版辅助阅读。

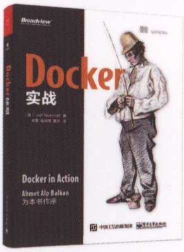
好书力荐

Spring Cloud
微服务实战

Spring MVC Cookbook
Spring MVC实战

亿级流量
网站架构核心技术
跟开涛学搭建高可用高并发系统


Mastering Microservices with Java
Java微服务


Docker
实战
Docker in Action
Ahmet Alp Balkan
为本书作序

Broadview®
www.broadview.com.cn
博文视点 · IT出版旗舰品牌
技术凝聚实力 · 专业创新出版

Spring Cloud与Docker
微服务架构实战

Spring Cloud 提供了构建分布式系统所需的“全家桶”，本书初稿完成后，我第一时间拿到了稿件，从零开始学习了 Spring Cloud。如果你想从零开始搭建一套分布式系统，《Spring Cloud 与 Docker 微服务架构实战》可以作为你的领路者，带你进入 Spring Cloud 的世界。

张开涛 《亿级流量网站架构核心技术》 作者

我认识的周立是一个对技术非常执着的“技者”，对 Spring Cloud 技术栈钻研得非常深入。这本书写得也非常实用，通过不同的角度来介绍 Spring Cloud，加入了很多实战的例子，值得一读。

曹祖鹏 千米网首席架构师

《Spring Cloud 与 Docker 微服务架构实战》从微服务设计原则和理念出发，详细说明了如何通过Spring Cloud及Docker建立高效可用的微服务解决方案，并对Spring Cloud的架构及组件、容器镜像的制作与编排进行了逐一讲解，具备较强的实战指导意义。本书能够帮助技术人员快速了解和应用微服务，通过技术的变革与提升帮助业务适应市场的快速变化，从而提升技术的价值。

廖俊杰 广发银行IDC团队负责

《Spring Cloud 与 Docker 微服务架构实战》这本书中，作者由浅入深地对 Spring Cloud 的主要常用组件进行了案例剖析和精彩讲解，让读者能快速上手，快速搭建基于 Spring Cloud 的微服务应用。

许进（xujin.org） Spring Cloud中国社区创始人，中间件高级研发工程师

博文视点Broadview

新浪微博
weibo.com
@博文视点Broadview

上架建议：Java微服务

ISBN 978-7-121-31271-7

9 787121 312717 >

定价：69.00元

PHEI

策划编辑：张春雨
责任编辑：徐津平
封面设计：王　乐